Sistemas Informáticos
Curso 2004-2005

---

# *Ajuste del planificador de Linux para procesadores con "Hyper-Threading"*


Edgardo Mejía Roa
David Rodrigo Ruiz
Javier Setoain Rodrigo

Dirigido por:
Prof. Manuel Prieto Matías y Luis Piñuel Moreno
Dpto. Arquitectura de Computadores y Automática


---


Facultad de Informática
Universidad Complutense de Madrid

# Prefacio

En este proyecto hemos abordado la sintonización del Kernel Linux para procesadores con tecnología *Hyper-Threading*, centrando nuestros esfuerzos en el diseño y desarrollo de un *planificador de procesos simbiótico*.

Por *software simbiótico* [16] entendemos aquél que es capaz de adaptarse dinámicamente para ajustar el escenario de ejecución con los requerimientos del sistema (ahorro de consumo, rendimiento, calidad de servicio...).

Para el desarrollo de dicho planificador, al que hemos bautizado con el acrónimo de *HTAS*, nos hemos basado en el uso de los contadores hardware de la arquitectura IA-32 (4). Mediante dichos contadores hemos podido detectar situaciones en las que *Hyper-Threading* puede comprometer el rendimiento.

Nos gustaría destacar que, aunque existen diversas propuestas basadas en estudios mediante simulación [15], no conocemos ninguna implementación de un planificador con estas características en sistemas reales [16]

We have addressed in this project the tuning of the Linux Kernel in *Hyper-Threading*-enabled processors, focusing on the design and implementation of a *symbiotic process scheduler*.

*Symbiotic software* is able to be dynamically adapted to match the execution scenario and the system policy (power saving, performance, quality of service...).

The symbiotic scheduler developed in this project, which we have dubbed as *HTAS*, is based on the employment of the IA-32 hardware performance counters (4) , which have help us to detect possible performance bottlenecks caused by *Hyper-Threading*.

We would like to remark that, despite being proposed by some authors in simulation studies [15], to the best of our knowledge any similar scheduler has been yet implemented on real systems [16].

# Lista de palabras

planificador, kernel, Linux, simbiótico, Hyper-Threading, SMT, contadores hardware, sistemas operativos, consumo de energía

Edgardo Mejía, David Rodrigo y Javier Setoain, autorizamos a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, la memoria derrollada en este proyecto.

Madrid, a 30 de Junio de 2005

Fdo.Edgardo Mejía  Fdo.David Rodrigo  Fdo.Javier Setoain

# Agradecimientos

Queremos dar las gracias a Nacho Gómez y a Christian Tenllado por su inestimable ayuda con el proyecto, a Luis Piñuel y Manuel Prieto, nuestros directores de proyecto, y a Enrique de la Torre, por aguantarnos durante todo el año. Muchas gracias a todos :-)

# Índice general

# Índice de figuras

# Índice de cuadros

# Capítulo 1

# Introducción a la tecnología *Multithreading*

Las dificultades que se están hallando para la explotación del paralelismo a nivel de instrucción, ha dado lugar a nuevos diseños que permiten un aprovechamiento más eficiente del creciente número de transistores. De entre ellas, destacaremos dos:

- *Multiprocesadores-en-un-chip* (MoC) [2]. En este tipo de diseños se aprovecha la mayor escala de integración para incluir más de un procesador (*core*) por circuito integrado. No existe un consenso sobre como debe ser la red que conecte dichos cores, barajándose alternativas muy diversas (conexiones externas, *crossbars*, anillos, mallas, buses...). Es habitual, además, que junto a los procesadores y sus correspondientes niveles de memoria cache (algunos de ellos compartidos) se integre algunos controladores complejos (coherencia, encaminamiento, DRAM...) y enlaces con un ancho de banda elevado que pueden conectarse directamente a otros chips (procesadores o conmutadores de expansión). De este modo se consigue un importante abaratamiento en la construcción de sistemas de mayor escala.

- *Procesadores Multithreading* [3]. Los procesadores multi-hilo (*multithreaded* o *multithreading*) intentan incrementar el rendimiento mediante la ejecución simultánea de varios hilos. Entre las diferentes alternativas destaca el *Multithreading* Simultáneo (SMT) [4] [5]. Los procesadores SMT tienen como base un superescalar fuera de orden capaz de ejecutar varias instrucciones por ciclo. Con cambios arquitectónicos relativamente poco importantes, estos procesadores se puede modificar apara lanzar simultáneamente instrucciones pertenecientes a varios flujos de

ejecución (correspondientes a un único proceso o a procesos independientes).

- *Chip-Multi-Thread* (CMT) [6]. En las últimas generaciones de procesadores ha comenzado a combinarse los dos paradigmas anteriores, es decir, se integran varios cores multithreading en un mismo chip.

La mayoría de los fabricantes han empezado a incorporar estas características en sus respectivas gamas de productos, y es muy probable que en un futuro próximo estos niveles de paralelismo adicionales sean tan habituales como hoy en día puede resultar el paralelismo a nivel de instrucción.

El ejemplo más representativo de procesador con SMT es el Intel Pentium 4 [1]. Entre los que integran MoC podemos citar el IBM Power4 [9], el AMD Opteron Dual-Core [10] y el SUN UltraSPARC IV+ [10]. Las novedades más recientes pertenecen sin embargo al paradigma CMT. Entre otros ejemplos podemos citar el Intel Pentium D [10], el Intel Montecito [11], el IBM Power5 [13], Cell [14], un MoC heterogéneo desarrollado conjuntamente por IBM-Sony-Toshiba, donde se incluye un core PowerPC multithreaded y el SUN Niagara [6], la apuesta de SUN para el mercado de servidores, donde se integran 8 cores *multithreading* (4 *threads* por *core*) en un mismo chip.

## 1.1. Arquitecturas *Multithreading*

Las arquitecturas *multithreading* aparecen como consecuencia de las limitaciones que imponen las dependencias de datos y la alta latencia de memoria en los procesadores convencionales. Se caracterizan por mantener varios flujos de instrucciones ejecutándose a la vez dentro del procesador. Dichos flujos pueden ser de una misma aplicación, con lo que se reduciría su tiempo de ejecución, o de aplicaciones diferentes, con lo que se aumentaría la productividad (el *throughput*) del sistema.

Bajo esta concepción genérica, pueden encontrarse diferentes propuestas. Las tres alternativas más populares que han sido exploradas hasta la fecha son el *multithreading* de grano fino (FMT: *fine-grain multithreading*), el *multithreading* de grano grueso (BMT: *block multithreading*) y el *multithreading* simultáneo (SMT: *simultaneous multithreading*) [18], [3].

### 1.1.1. Multithreaded de Grano Fino (FMT)

En las arquitecturas con *multithreading* de grano fino (también conocido como *interleaved multithreading* o *multithreading* horizontal), se produce un cambio de contexto en cada ciclo, utilizando habitualmente políticas de planificación justas tipo *round-robin*. De este modo, aunque no se llegan a emitir en el mismo ciclo instrucciones de flujos o *threads* diferentes, sí conviven dentro del *pipeline*. El objetivo que se persigue con este tipo de diseños es el aumento de la productividad, ocultando las operaciones de latencia alta, como los accesos a memoria, con trabajo útil de otros *threads*.

Los cambios de contexto se llevan a cabo sin ninguna sobrecarga adicional, ya que los procesadores que implementan esta técnica disponen de tantos contextos hardware como número de *threads* soporten, evitándose de esta forma la necesidad de salvar dichos contextos. Por otro lado, también es habitual que el lanzamiento de una nueva instrucción de un determinado *thread* no se realice hasta que la instrucción anterior de ese mismo *thread* haya sido retirada. Se simplifica de este modo el diseño del *pipeline*, ya que al eliminarse tanto los riesgos de control como los riegos de datos, no es necesario incluir mecanismos sofisticados de control ni *forwardings* de datos complejos. Las latencias de memoria pueden ocultarse impidiendo la planificación del correspondiente *thread* hasta que el acceso se haya completado [3].

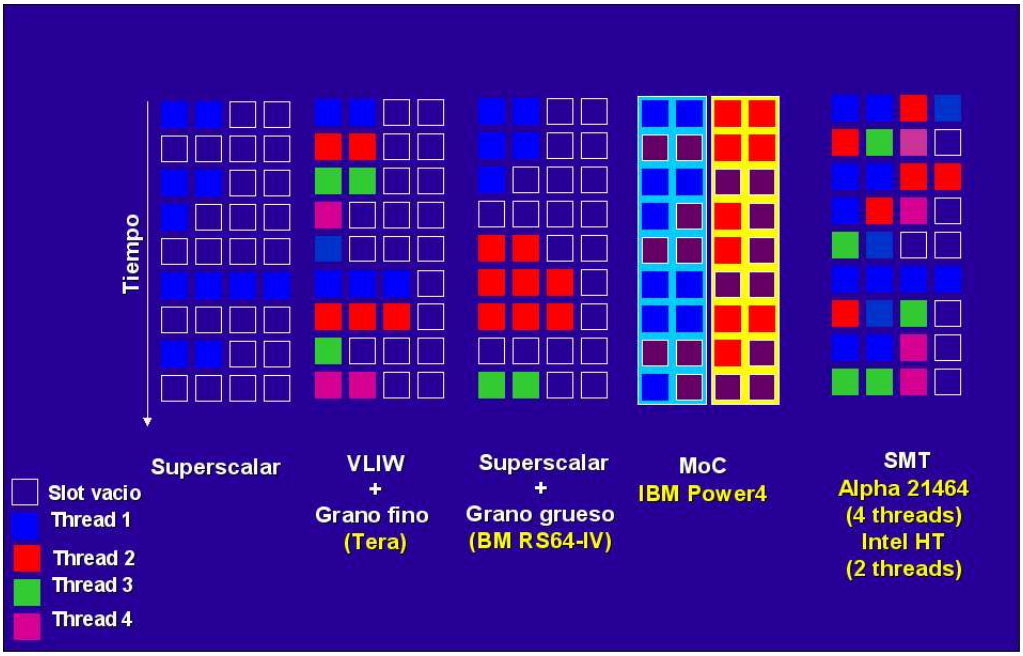


Figura 1.1: Paradigmas *multithreaded* propuestos
Ejecución de instrucciones en un procesador superescalar de ancho cuatro convencional (izquierda), un procesador VLIW (4 operaciones por instrucción) con *multithreading* de grano fino, un procesador superescalar con *multithreading* de grano grueso y un procesador superescalar con capacidad SMT (derecha). *Multithreading* de grano fino y de grano grueso ocultan latencias, SMT trata de aprovechar todo el ancho superescalar, ocultando latencias indirectamente.

### 1.1.2. Multithreaded de Grano Grueso (BMT)

En las arquitecturas con *multithreading* de grano grueso (también denominado *blocked multithreading*) los cambios de contexto no se producen automáticamente cada ciclo, sino que se producen ante la ocurrencia de eventos muy costosos que puedan provocar largas paradas del *pipeline*. El coste del cambio de contexto varía con las diferentes propuestas. En arquitecturas con suficientes recursos, el cambio de contexto puede ser inmediato, pero en arquitecturas donde sea necesario salvar el contexto de los *threads*, el coste puede ser de varios ciclos.

### 1.1.3. Multithreading Simultáneo (SMT)

El *multithreading* simultáneo es sin duda la alternativa *multithreaded* que más repercusión comercial y mediática ha despertado recientemente. La característica diferencial de este tipo de arquitecturas es que permiten el lanzamiento (*issue*) de instrucciones provenientes de diferentes flujos en el mismo ciclo sin necesidad de cambios de contexto [18] [3].

En la figura 1.1 se comparan los distintos paradigmas *multithreading* que se han propuesto. SMT puede considerarse como una evolución natural del las arquitecturas superescalares cuya intención fundamental es la conversión de paralelismo a nivel de *thread* (TLP: *Thread Level Parallelism*), en paralelismo a nivel de instrucción. Con ello se persigue el aprovechamiento del ancho superescalar, a menudo infrautilizado al carecer las aplicaciones del suficiente ILP, lo que redunda en una mayor utilización de los recursos de procesador. Indirectamente, se mitigan de este modo las ineficiencias producidas por operaciones con una latencia alta.

Las primeras propuestas surgieron dentro del mundo académico a mediados de la década de los 90 ([4] [8] [5]), despertando desde entonces un gran interés a nivel investigador. La consolidación del SMT en la industria no llega hasta el año 2002, con su introducción en los procesadores Intel Pentium 4 e Intel Xeon1 [1]. El Alpha 21464 (EV8) [7], aunque nunca llegó al mercado, estaba previsto que utilizase este paradigma y en la actualidad, también encontramos está técnica en el IBM Power5 [12], [13].

La figura 1.2 muestra el *pipeline* de ejecución básico de un procesador SMT. Partiendo de una arquitectura superescalar, las modificaciones más importantes se han de hacer fundamentalmente en la etapa de búsqueda de instrucciones, manteniéndose prácticamente inalterado el resto del procesador.

Con cargas de trabajo multiprogramadas incrementa la productividad del sistema, siempre y cuando no haya competencia de recursos entre los correspondientes threads. Sin embargo, si lo que se pretende es reducir el tiempo de ejecución de una aplicación, el nuevo reto que plantean este tipo de arquitecturas es como extraer el paralelismo a nivel de *thread*.

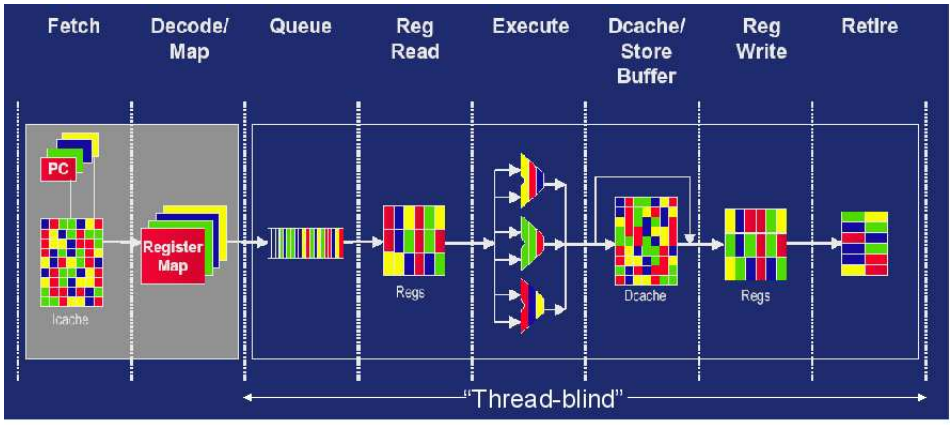


Figura 1.2: *Pipeline* del Alpha 21464
Básicamente se modifica sólo la etapa de *fetch*

## 1.2. *Hyper-Threading*: SMT en el Intel Pentium 4 y Xeon

Una de las primeras implementaciones comerciales de SMT la encontramos en los procesadores de la familia Xeon de Intel [1] . En sus descripciones, Intel suele representar a la tecnología *Hyper-Threading*, como una virtualización del procesador físico que permite interpretarlo como dos procesadores lógicos. Como se muestra en la figura 1.3, para ello se duplica la arquitectura del procesador (registros de propósito general, registros de control, y registros para el control de las interrupciones), manteniendo compartidos la mayor parte de los recursos (caches, unidades aritmético-lógicas, buses,...). Se proporciona así la ilusión al software de que existen dos procesadores independientes.

Los primeros sistemas con la tecnología *Hyper-Threading* datan de 1998. La incorporación del *Hyper-Threading* suponía un incremento del tamaño del chip de menos de un 5 %, mientras que la mejora del rendimiento variaba entre 16 y 28 % [1] según las aplicaciones.

### 1.2.1. Recursos duplicados, repartidos y compartidos

La opción tomada por Intel para la implementación del *multithreading* consiste en añadir el mínimo hardware posible. Los recursos del procesador, ahora serán utilizados por los dos *procesadores lógicos*. Algunos de estos recursos, debido a su pequeño tamaño y coste, serán duplicados. El problema

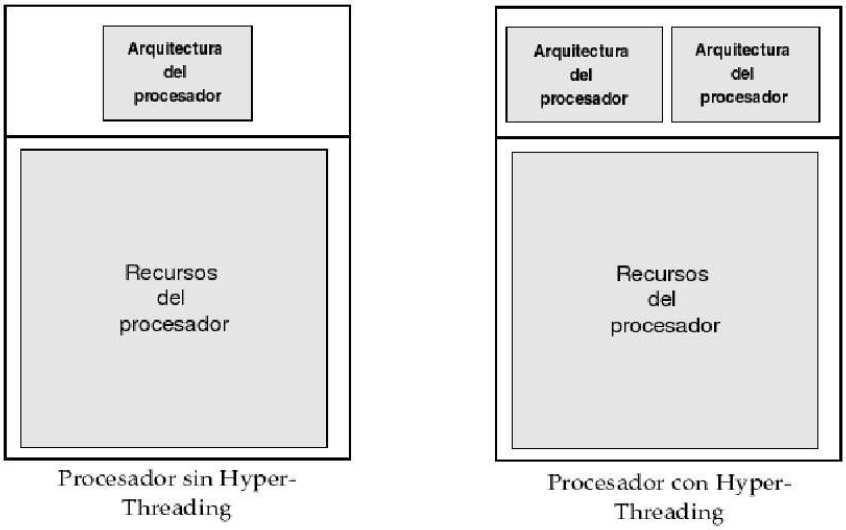


Figura 1.3: Procesador con *Hyper-Threading* (derecha) y sin él (izquierda)

viene dado por los recursos que se reparten y los compartidos:

- Los recursos repartidos, tienen limitado a cada *procesador lógico* el uso en tiempo (cada ciclo accede un procesador lógico) o espacio (el recurso se divide entre los procesadores lógicos). Normalmente esto se hace con recursos cuya duplicación ya supone cierto coste, y supone una limitación en el rendimiento. El reparto de estos recursos es a partes iguales, y cuando el procesador funciona en modo monotarea, se produce una combinación de recursos para que el procesador disponga de todo el hardware [1].

- Los recursos compartidos, no conllevan equidad en el reparto. Un ejemplo de recurso compartido es la memoria cache de nivel 1 y 2, que no se duplica al ser un recurso de considerable tamaño y por tanto caro. Por consiguiente, un buen uso de estos recursos será crítico para el rendimiento.

# Capítulo 2

# El Planificador de Linux 2.6.12

Linux es un sistema muy complejo, compuesto de muchos componentes divididos en capas, relacionados entre sí. Estos componentes se comunican, se sincronizan y funcionan de manera concurrente, lo que hace que un sistema ya de por sí complejo por su propia extensión, se convierta en algo en lo que las consecuencias de pequeños cambios en sistemas clave se hagan difíciles de predecir.

Dentro del núcleo existe una parte central que articula los mecanismos principales del sistema, que llamamos *los subsistemas del kernel*. Estos subsistemas incluyen el gestor de memoria, el sistema de ficheros virtual, el subsistema de comunicación entre procesos, el subsistema de red y el planificador. En la figura 2.1 se puede ver en qué punto encaja cada subsistema y las capas que tiene. Como se puede apreciar, en el centro de todos los subsistemas se encuentra planificador.

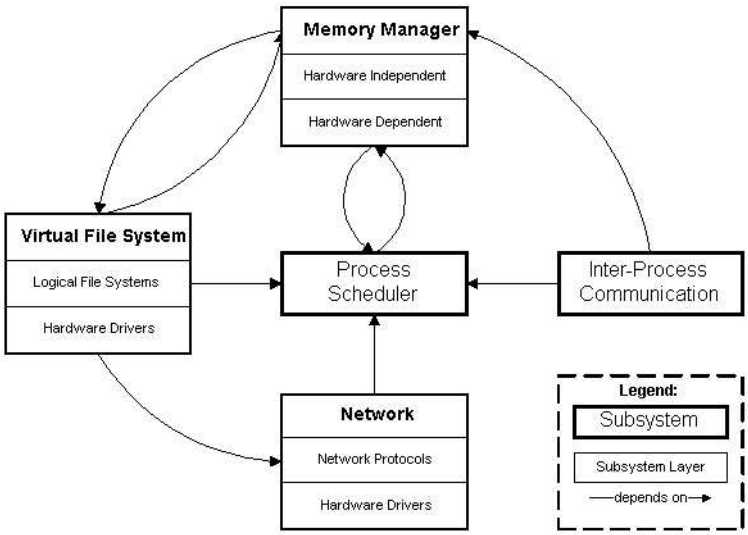


Figura 2.1: Subsistemas del kernel

Es decir, el planificador es un subsistema del que dependen todos los demás, lo cual implica dos cosas:

1. Es crítico en rendimiento. Una pérdida de rendimiento mínima la notarán todos los subsistemas y repercutirá en el núcleo entero de diversas maneras, lo cual implicará una severa degradación de todo el sistema operativo.

2. Introducir cambios es extremadamente complejo. Al depender todos los subsistemas de él, cualquier modificación está afectando de forma indirecta a todos los demás subsistemas, y hay que tener en cuenta una infinidad de efectos colaterales en los que podría desembocar el cambio.

La misión del planificador de un sistema operativo es decidir cómo, cuándo y dónde se ejecutarán las tareas que están corriendo en el sistema. Ésto implica, entre otras cosas, hacer listas de procesos en base a sus características, asignarles periodos de ejecución e intercambiar unas tareas por otras.

El planificador, en la rama de Linux 2.6.x, ha sido reescrito entero e incorpora grandes cambios respecto a versiones anteriores, las cuales han mejorado mucho sus capacidades, rendimiento y escalabilidad. Algunos de los cambios más significativos en el planificador, como la complejidad en $O(1)$ o la expropiación, son explicados en detalle a lo largo de este capítulo.

## 2.1. Estructura del Planificador

El planificador de Linux está implementado en `kernel/sched.c`. El algoritmo fue reescrito por completo en la versión 2.5 y, al contrario que en las versiones anteriores, fue diseñado para cumplir objetivos específicos:

- Complejidad $O(1)$: cada algoritmo en el nuevo planificador debe finalizar en tiempo "constante", independientemente del número de procesos que haya corriendo en el sistema.
- Escalabilidad "perfecta" en SMP: cada procesador debe tener su propia Cola de procesos (*runqueue*) y su *lock* de protección asociado. De esta manera es posible que de forma simultánea, dos tareas se despierten en distintas CPUs, sean planificadas y se hagan los correspondientes cambios de contexto.
- Afinidad SMP mejorada: el planificador debe intentar agrupar y mantener a los procesos en una CPU específica. Sólo se migrarán a otra CPU para conseguir un mejor equilibrado de carga. Así, se impide que haya tareas que estén continuamente siendo migradas de una Cola de procesos a otra.
- Eficiencia SMP: ninguna CPU debe estar inactiva si hay tareas que ejecutar.
- Planificación por lotes: *timeslices* largos y política *Round-Robin*. Se aplica a las tareas con menor prioridad, es decir, con un valor de *nice* positivo.
- Ejecutar los procesos hijos recién creados antes que sus procesos padres.
- Buen rendimiento con tareas interactivas: incluso con una carga considerable, el sistema debería reaccionar y planificar las tareas interactivas inmediatamente. Esto beneficia enormemente a sistemas tipo *desktop*.
- Optimizado para el caso de una o dos tareas en ejecución.
- Reparto justo de los *timeslices*: ningún proceso debe sufrir inanición ni tener de manera injustificada un *timeslice* demasiado alto.

A continuación veremos las principales estructuras de datos presentes en el planificador:

### 2.1.1. Procesos

Toda la información relativa a los procesos se guarda en su *Descriptor de Proceso*. Esta información incluye ficheros abiertos, los datos de su espacio de memoria, señales pendientes, estado del proceso, etc. En Linux, la información del descriptor de procesos se guarda en la estructura `task_t` que está definida en `include/linux/sched.h`.

A diferencia de otros sistemas operativos, en Linux son totalmente equivalentes los conceptos de *procesos* e *hilos de ejecución* (*threads*). Otra peculiaridad de Linux, es que emplea el nombre de *tareas* (*tasks*) para referirse a los procesos. Por tanto se utilizarán ambos términos de manera indistinta.

#### Estructura de los procesos

La estructura `task_t` posee muchos campos, pero los más importantes, en lo que a la planificación se refiere, son los siguientes:

```
typedef struct task_struct task_t;

struct task_struct {
    prio_array_t *array;             // Array de prioridades en el que está
    struct list_head run_list;       // Puntero a la siguiente tarea en el Array de prioridades
    int prio;                        // Prioridad efectiva del proceso
    int static_prio;                 // Prioridad estática del proceso
    unsigned int time_slice;         // Tiempo de ejecución
    unsigned long sleep_avg;         // Tiempo medio que pasa suspendido
    volatile long state;             // Estado del proceso
    int activated;                   // Estado en el que estaba suspendido
    unsigned long rt_priority;       // Prioridad en tiempo real
    unsigned long policy;            // Esquema de planificación
    cpumask_t cpus_allowed;          // CPUs permitidas para este proceso
    spinlock_t switch_lock;          // Lock para los cambios de contexto.
    struct thread_info *thread_info; // Acceso eficiente al Descriptor de Proceso.
    struct sched_info sched_info;    // Estadísticas a nivel de proceso.
};
```

A continuación se describen algunos de estos campos:

- `prio_array_t *array`: arrays que albergan a todos los procesos clasificándolos por su prioridad.
- `struct list_head run_list`: lista de procesos asociada al nivel de prioridad de la tarea.

  Los Arrays de Prioridad y sus listas de procesos se explicarán en la sección 2.1.3, página 21.
- `unsigned long rt_priority`: es la representación que ve el usuario de los niveles de prioridad de los procesos de tiempo real.

- `unsigned long policy`: es el esquema de planificación empleado para este proceso.

  Tanto este, como el campo `rt_priority` se explican en la sección 2.4.4, página 47.

- `int static_prio`: Prioridad Estática. Es la prioridad elegida por el usuario para los procesos que no son de tiempo real (los procesos de usuario).

- `int prio`: es la prioridad efectiva del proceso, tanto si es de tiempo real, como si es de usuario.

- `unsigned long sleep_avg`: tiempo medio que el proceso pasa suspendido (*sleep average*). Es una heurística empleada por el planificador para medir la "interactividad" de un proceso, y así, ajustar su prioridad.

  Las prioridades de los procesos de usuario se explican en la sección 2.4.1, página 38.

- `volatile long state`: estado del proceso. Se explica más adelante.

- `int activated`: este campo es empleado cuando se despierta a un proceso. En función del estado en el que estuvo suspendido, se le da al campo un valor que representa su "grado de interactividad". Posteriormente, se modificará su *sleep average* según este valor. Para más detalles, ver la sección 2.5.2 (página 50).

- `int time_slice`: tiempo de ejecución del proceso. Este valor es elegido en función de su prioridad Estática.

- `cpumask_t cpus_allowed`: máscara que indica los procesadores en los que se puede ejecutar el proceso. Se emplea en los equilibrados de carga (ver sección 2.3, página 33).

- `spinlock_t switch_lock`: *lock* empleado en algunas arquitecturas durante los Cambios de contexto. Ver sección 2.5.1 (página 49).

- `struct thread_info *thread_info`: estructura empleada para acceder de manera eficiente al descriptor de procesos. Se explica más adelante.

- `struct sched_info sched_info`: estructura utilizada en la recolección de estadísticas a nivel de proceso. Se detallan en la sección 2.5.5 (página 55).

### Estados de los procesos

El estado de los procesos es descrito por el campo `state` de la estructura `task_t`. Algunos de los estados de los procesos son los siguientes:

- `TASK_RUNNING`: el proceso es ejecutable y por tanto se encuentra en el Array de Prioridad, ya sea en ejecución, o esperando su turno. El valor por defecto de esta constante es 0.

- `TASK_INTERRUPTIBLE`: el proceso está suspendido esperando a que se cumpla alguna condición. Cuando esto ocurra, el proceso será despertado y puesto en estado `TASK_RUNNING`. Sin embargo, también puede ser despertado de forma prematura si recibe alguna señal (por ejemplo, `SIGTERM`). Los procesos interactivos suelen estar suspendidos en este estado.

- `TASK_UNINTERRUPTIBLE`: igual que el caso anterior, salvo que el proceso no será despertado si recibe alguna señal. Para más detalles, ver la sección 2.5.2 (página 50).

- `TASK_TRACED`: el proceso está siendo depurado.

- `TASK_STOPPED`: la ejecución del proceso ha sido detenida. En este caso el proceso ya no se ejecuta ni puede ser elegido por el planificador para ejecución. Esto ocurre cuando la tarea recibe las señales `SIGSTOP`, `SIGSTP`, `SIGTTIN` o `SIGTTOU`, o bien, si se recibe cualquier señal mientras está siendo depurada.

- `EXIT_ZOMBIE`: el proceso ha acabado, pero su descriptor sigue en memoria por si el padre necesita alguna información. Dicho descriptor será eliminado cuando el padre ejecute la llamada al sistema `wait4()`.

- `EXIT_DEAD`: el proceso ha acabado y se ha eliminado su descriptor.

El mecanismo más empleado para cambiar el estado de un proceso es utilizando las macros `set_task_state(task,state)` y `set_current_state(state)`. La diferencia es que la segunda sólo cambia el estado del proceso `current` (el que está actualmente en ejecución). En sistemas SMP, además, puede establecer una barrera para sincronizar a las tareas del resto de procesadores.

Las transiciones entre estados de los procesos, y sus causas, se pueden observar en la figura 2.2 ([20]).

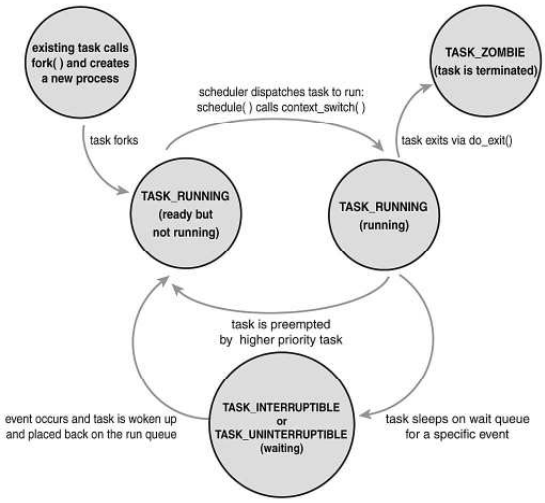


Figura 2.2: Estados de los procesos.

### Accediendo al Descriptor de Proceso

En versiones de Linux anteriores a la 2.6, el descriptor de procesos (`task_t`) se almacenaba al final de la pila de sistema (*kernel stack*) de cada proceso, de tal manera que en arquitecturas con pocos registros, como la x86, para acceder a este descriptor se empleaba el puntero a la cima de la pila como registro base.

En esta nueva versión de Linux, los descriptores de procesos son creados de manera dinámica utilizando el *slab allocator* ([20]). Para acceder a ellos, se ha creado una nueva estructura al final de la pila de sistema (donde estába antes el descriptor de procesos) denominada `thread_info`, en la cual, entre otros campos, hay un puntero al descriptor (campo `task`).

En el código del planificador, es muy común acceder al descriptor del proceso que está actualmente en ejecución, por ello existe una macro denominada `current` para acceder a este descriptor de manera más eficiente. Esta macro esta implementada para cada arquitectura en `include/asm-*/current.h`. En algunas de ellas, como la x86, invoca a la función `current_thread_info()` (implementada en ensamblador) para obtener el `thread_info` del proceso, y luego su descriptor. En la figura 2.3 ([20]) se puede observar la estructura `thread_info` situada en la pila de sistema del proceso.

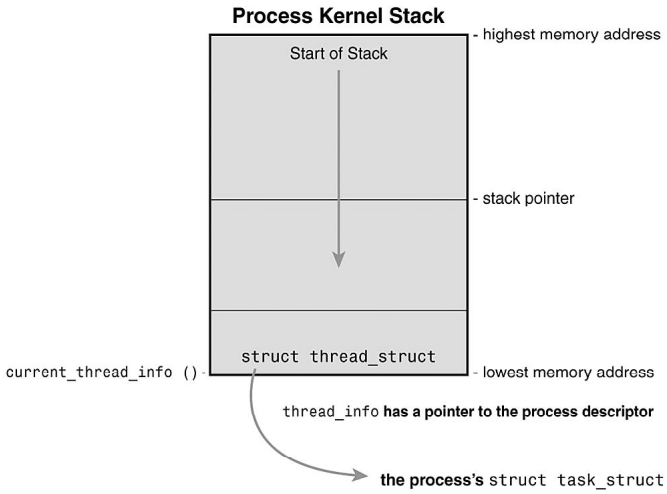


Figura 2.3: Descriptor de proceso en la pila de sistema.

### Inicio de un proceso

Tradicionalmente, al crear un proceso hijo con la llamada al sistema `fork()`, casi todos los recursos que poseía el proceso padre eran duplicados y copiados al hijo. De manera que sólo se diferenciaban en el PID, el PPID (*parent's PID*) y en algunas estadísticas y recursos tales como las señales pendientes. Esta aproximación es bastante ineficiente, sobre todo si lo primero que ejecuta el proceso hijo es una llamada a `exec()` la cual carga un nuevo programa en el espacio de memoria, convirtiendo así, en innecesaria y en una pérdida de tiempo, la copia del espacio de memoria del proceso padre (que fácilmente puede alcanzar los 10 Mbytes).

Actualmente, esta llamada al sistema emplea un mecanismo denominado *Copy-On-Write* (*COW*), cuyo objetivo es el de retrasar, o incluso evitar, la copia innecesaria de recursos a la hora de crear un proceso hijo. Con esta técnica, en lugar de duplicar todo el espacio de memoria, el proceso padre y el hijo comparten una misma y única copia. De esta forma, los datos que contiene sólo se duplicarán si uno de los procesos intenta modificar alguna de las páginas de ese espacio de memora.

Por tanto, si el proceso hijo ejecuta un `exec()` justo después del `fork()`, ninguno de los datos del espacio de memoria del padre habrán sido duplicados. La única carga producida por `fork()` es la copia de las tablas de páginas del padre y la creación de un nuevo descriptor de procesos para el hijo.

Sin embargo, se puede optimizar aún más si el kernel despierta y ejecuta al proceso hijo *antes* de continuar con el padre (en previsión de que el hijo ejecutará inmediatamente un `exec()`). Si se hace al revés, se corre el riesgo de que el padre comience a hacer modificaciones en las páginas del espacio de memoria, y que por tanto, estas se dupliquen al hijo (innecesariamente, puesto que este último ejecutará otro programa). Esta acción es llevada a cabo en la función `wake_up_new_task()` y se denomina *child-run-first*.

### 2.1.2. Colas de procesos

La principal estructura de datos del planificador es la *Cola de Procesos* o *Runqueue* y se trata de la lista de los procesos ejecutables en cada procesador. Existe una cola para cada procesador y cada proceso ejecutable sólo puede estar en una cola a la vez. Está definida como `runqueue_t` en `kernel/sched.c`, aunque podría haberse hecho en `include/linux/sched.h`, pero lo que se pretende es aislar al código del planificador y mostrar sólo una interfaz reducida al resto del kernel.

#### Estructura de la Cola de Procesos

La estructura de la cola de procesos es la siguiente ([20]):

```
typedef struct runqueue runqueue_t;

struct runqueue {
    spinlock_t lock;                        // Cerrojo para proteger esta cola
    unsigned long nr_running;               // Número de tareas ejecutables
    unsigned long cpu_load;                 // Carga del procesador
    unsigned long long nr_switches;         // Número de cambios de contextos
    unsigned long expired_timestamp;        // Tiempo desde el último intercambio de arrays
    unsigned long nr_uninterruptible;       // Número de tareas en estado TASK_UNINTERRUPTIBLE
    unsigned long long timestamp_last_tick; // Marca de tiempo del último 'tick' del planificador
    task_t *curr;                           // Proceso en ejecución
    task_t *idle;                           // Proceso 'ocioso' de esta cola
    struct mm_struct *prev_mm;              // Mapa de memoria virtual del proceso anterior
    prio_array_t *active;                   // Array de prioridad de tareas activas
    prio_array_t *expired;                  // Array de prioridad de taras expiradas
    prio_array_t arrays[2];                 // Arrays de prioridad
    int best_expired_prio;                  // La mayor de las prioridades de las tareas expiradas
    atomic_t nr_iowait;                     // Número de tareas esperando E/S
    struct sched_domain *sd;                // Dominio de planificación de esta cola
    int active_balance;                     // Indica si la cola necesita equilibrado
    int push_cpu;                           // CPU a la que se envían tareas durante un equilibrado
    task_t *migration_thread;               // Proceso para la migración de tareas
    struct list_head migration_queue;       // Lista de tareas a migrar durante un equilibrado
};
```

A continuación se explica cada campo ([21]):

- `spinlock_t lock`: cerrojo para proteger el acceso a la cola. De esta manera, sólo una tarea puede modificarla. El uso de este campo se explicará con detalle en la sección 2.1.2 de la página 20.

- `unsigned long nr_running`: número de procesos ejecutables que están en la cola.

- `unsigned long cpu_load`: carga de trabajo del procesador. Esta carga se calcula y se actualiza en el método `rebalance_tick()` haciendo la media entre la carga anterior y la actual.

- `unsigned long long nr_switches`: número de cambios de contexto que han ocurrido desde la creación de la cola al iniciarse el sistema. Actualmente sólo se utiliza al mostrar estadísticas en el sistema de ficheros `/proc`.

- `unsigned long expired_timestamp`: tiempo transcurrido desde la última vez que se intercambiaron los Arrays de prioridad.

- `unsigned long nr_uninterruptible`: número de tareas que se encuentran en el estado `TASK_UNINTERRUPTIBLE`.

- `unsigned long long timestamp_last_tick`: marca de tiempo del último *tick* del planificador. Se utiliza principalmente en la macro `task_hot()` para determinar si un proceso es *cache hot*, es decir, si ha transcurrido muy "poco" tiempo y por tanto, es muy probable que en la cache de la CPU todavía se encuentren datos válidos del proceso.

- `task_t *curr`: es el proceso que actualmente se está ejecutando en este procesador.

- `task_t *idle`: es el proceso "ocioso" de esta cola, es decir, el proceso que se ejecuta cuando no hay otras tareas a ejecutar.

- `struct mm_struct *prev_mm`: mapa de la Memoria Virtual de la tarea que estuvo ejecutándose anteriormente. Se utiliza para una gestión eficiente de la memoria virtual.

- `prio_array_t *active`: contiene las tareas que no han agotado aún todo su tiempo de ejecución (*timeslice*).

- `prio_array_t *expired`: contiene las tareas que ya han agotado todo su *timeslice*.

- `prio_array_t arrays[2]`: arrays de prioridad. Estas estructuras se explicarán con detalle en la sección 2.1.3 en la página 21.
- `int best_expired_prio`: la mayor de las prioridades (la del mínimo valor) de las tareas expiradas. Se utiliza en la macro `EXPIRED_STARVING()` para determinar si existe una tarea con mayor prioridad que la actual en el array de tareas expiradas. De esta manera, si la tarea actual es interactiva, no se reinserta en el array de las activas sino en el de las expiradas y así se evita la inanición de las tareas expiradas. Esto se explica con detalle en la sección 2.4.2 en la página 41.
- `atomic_t nr_iowait`: número de procesos esperando en operaciones de E/S. Utilizado para las estadísticas del kernel.
- `struct sched_domain *sd`: dominio de planificación para esta cola. Básicamente es el grupo de CPUs entre los cuales se hace el equilibrado activo. Se explicará con detalle en la sección 2.1.4 en la página 24.
- `int active_balance`: flag empleado en la migración de tareas para determinar si la cola debe ser equilibrada, es decir, si está considerablemente más llena que el resto.
- `int push_cpu`: procesador al que son enviadas tareas durante el equilibrado.
- `task_t *migration_thread`: proceso encargado de migrar tareas a otro procesador durante un equilibrado de carga.
- `struct list_head migration_queue`: lista de tareas que deben ser migradas a otra CPU.

Existen otros campos adicionales que sólo se utilizan para las *Estadísticas del Planificador* (ver sección 2.5.5, página 55). Únicamente se compilan si está definida la variable `CONFIG_SCHEDSTATS`.

Dado que las colas de procesos son la principal estructura de datos del planificador, existen una serie de macros definidas en `kernel/sched.c` para manejarlas:

- `cpu_rq(cpu)`: devuelve un puntero a la cola de procesos del procesador `cpu`.
- `this_rq()`: devuelve un puntero a la *runqueue* del procesador actual.
- `task_rq(p)`: devuelve un puntero a la cola donde está el proceso `p`.

### Bloqueos

Como se explica en la sección 2.4.3 de la página 43, el nuevo planificador de Linux es *completamente expropiativo*, y por tanto puede seleccionar a una nueva tarea tanto si la anterior está en modo usuario, como si está en modo núcleo. Esto obliga a proporcionar métodos que controlen el acceso concurrente a los recursos del planificador. Un ejemplo de ello son las *runqueues*, que poseen mecanismos de bloqueo individuales ya que estas pueden ser modificadas, no solamente por el procesador al que están asociadas, sino también por otros procesadores (por ejemplo, durante un equilibrado de carga).

Estos mecanismos de bloqueo están implementados mediante un cerrojo (*lock*) que hay que adquirir antes de poder leer o escribir en la estructura de la cola y que debe ser liberado al finalizar la operación. Todo esto se puede realizar con las siguientes funciones:

- `task_rq_lock(p, flags)`: bloquea y devuelve un puntero a la cola en la que el proceso `p` se está ejecutando. Además deshabilita las interrupciones guardando el estado en `flags`.
- `task_rq_unlock(rq, flags)`: desbloquea la cola `rq` y habilita las interrupciones con el estado guardado en `flags`.
- `this_rq_lock()`: bloquea y devuelve un puntero a la cola asociada al procesador actual. También deshabilita las interrupciones, pero sin guardar el estado.
- `rq_unlock(rq)`: desbloquea la cola `rq` y habilita las interrupciones con el estado por defecto.

Otra posibilidad, es acceder directamente al campo `spinlock_t lock` de la estructura `runqueue_t`, y utilizar los métodos `spin_lock(&rq->lock)` y `spin_unlock(&rq->lock)` para bloquearla y desbloquearla respectivamente. Si además de deshabilitar la expropiación del planificador también se quiere deshabilitar las interrupciones, entonces deben emplearse las funciones `spin_lock_irqsave()` y `spin_unlock_irqrestore()`.

En el caso de querer bloquear varias *runqueues*, es necesario hacerlo en orden ascendente de direcciones para evitar interbloqueos. Por ejemplo ([20]):

```
/* Para bloquear las colas... */
if ( rq1 < rq2 ) {
    spin_lock(&rq1->lock);
    spin_lock(&rq2->lock);
}
else {
    spin_lock(&rq2->lock);
    spin_lock(&rq1->lock);
}

/* [Sección crítica] */

/* Para desbloquearlas... */
spin_unlock(&rq1->lock);
spin_unlock(&rq2->lock);
```

Aunque esto se puede realizar de manera automática con las funciones `double_rq_lock(rq1, rq2)` y `double_rq_unlock(rq1, rq2)`.

### 2.1.3. Arrays de prioridad

Los Arrays de Prioridad son las estructuras de datos claves que han permitido obtener un planificador con coste $O(1)$. Gracias a ellos, se puede conseguir transiciones de época[1] en tiempo "constante", independientemente del número de procesos en ejecución. Además, permiten encontrar de manera muy eficiente, cuál es el siguiente proceso a ejecutar.

En cada cola de procesos existen dos de estos arrays, el de *Tareas Activas* y el de *Expiradas*. Las tareas activas son aquellas que no han consumido todo su *timeslice*, mientras que las expiradas, son aquellas que ya lo han agotado. Cada array de prioridad contiene una lista de procesos por cada nivel de prioridad y un *Bitmap de Prioridad* para encontrar eficientemente la tarea con mayor prioridad (que será la siguiente a ejecutar). Esto se puede ver en la figura 2.4 ([25]).

La estructura de estos arrays, definida en `kernel/sched.c` como `prio_array_t` es la siguiente ([20]):

```
typedef struct prio_array prio_array_t;

struct prio_array {
    unsigned int nr_active;             /* Número de procesos en el array. */
    unsigned long bitmap[BITMAP_SIZE];  /* Bitmap de prioridad. */
    struct list_head queue[MAX_PRIO];   /* Array de colas de procesos. */
};
```

[1] Una *época* es el tiempo transcurrido desde el instante en que todos los procesos obtienen un nuevo *timeslice* (inicio de época) y el momento en el que todos los procesos consumen dicho *timeslice* (fin de época).

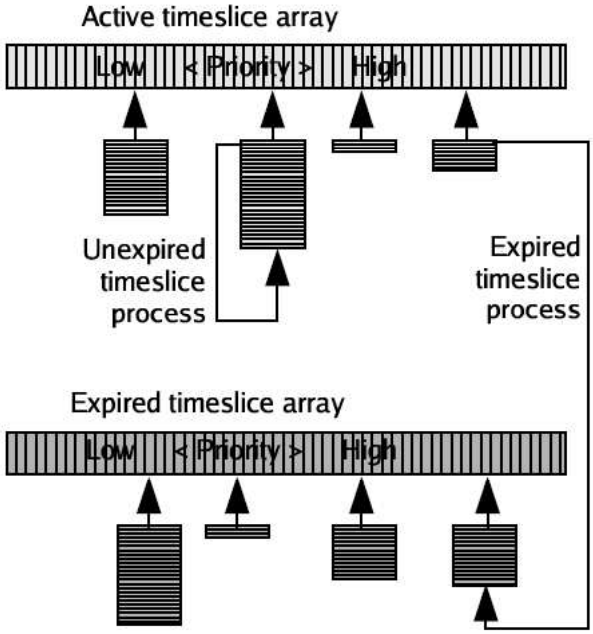


Figura 2.4: Arrays de Prioridad.

A continuación se explica cada campo ([21]):

- `unsigned int nr_active`: es el número de procesos que se encuentran en el array.
- `struct list_head queue[MAX_PRIO]`: es un array de listas enlazadas de procesos (ver figura 2.4, [25]). Cada lista se corresponde con un nivel de prioridad (por defecto, `MAX_PRIO` es 140), de manera que cada proceso se inserta según ese nivel. Por ejemplo, un proceso con prioridad 7 se insertará al final de la lista que hay en `queue[7]`. Por tanto, los procesos de mayor prioridad, serán siempre los de la lista que tenga el índice más bajo en este array. En estas listas, los procesos están enlazados entre sí, a través del campo `run_list` de su estructura `task_t`. (ver figura 2.5, [20]).
- `unsigned long bitmap[BITMAP_SIZE]`: array de `unsigned long` cuyos bits representan los niveles de prioridad de los procesos. `BITMAP_SIZE` es el número de `unsigned long` necesarios para que haya un total de `MAX_PRIO` bits. Por ejemplo, con 140 niveles de prioridad y 32 bits por variable, `bitmap[]` debe ser un array de tamaño 5 (160 bits).

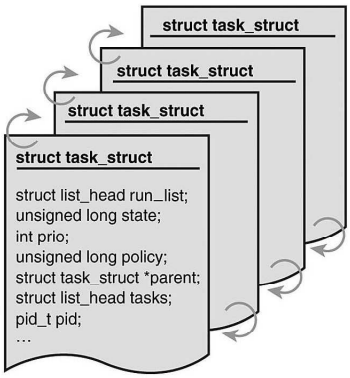


Figura 2.5: Lista de procesos para un nivel de prioridad dado.

Inicialmente, todos los bits están a 0. Cada vez que una tarea se inserta en una de las listas de `queue[]`, se establece a 1 el bit correspondiente al nivel de dicha tarea. Por ejemplo, al insertar un proceso con prioridad 7 en `queue[7]`, se establece a 1 el bit 7 de este campo. Para encontrar la tarea de mayor prioridad, basta con buscar el primer bit que valga 1 mediante la función `sched_find_first_bit()`. Con esto se evita tener que recorrer el array de listas, obteniendo así, un algoritmo de complejidad $O(1)$ en el número de procesos en ejecución. Esto se explica con detalle en la sección 2.2 (página 30).

Al comentar las *Prioridades de los procesos*, en la sección 2.4.1 (página 38), se explicará cómo el campo `prio`, de la estructura `task_t` de cada proceso, es el que almacena el nivel de prioridad, y por tanto, es el utilizado como índice en estos arrays.

De hecho, este es el campo utilizado por las funciones `enqueue_task()` y `dequeue_task()`, para insertar y eliminar los proceso en los Arrays de prioridad. Además, si este proceso es el primero en ser insertado, o el último en ser eliminado, estas funciones activan o desactivan (respectivamente) el correspondiente bit de `bitmap[]`.

### Recalculando los *timeslices*

Muchos sistemas operativos, incluyendo antiguas versiones de Linux, tienen una función explícita para calcular los *timeslices* de cada proceso al finalizar una *época* del planificador. Básicamente se trata de un bucle que recorre las

listas de tareas y calcula, para cada proceso, un nuevo tiempo de ejecución en función de su prioridad. Este método posee varias desventajas:

- Es un método de complejidad *lineal* ($O(n)$) en el número de procesos en ejecución.
- Es necesario bloquear la lista de tareas mientras se ejecuta el bucle, lo cual implica que debe detenerse el sistema por completo.
- El que esto ocurra de manera aleatoria, hace su utilización inadecuada para procesos de tiempo real.

El nuevo planificador resuelve estos problemas mediante el uso de los *Arrays de Prioridad*. Como se mencionó anteriormente, cada *Runqueue* (es decir, cada procesador) posee dos de estos arrays: el de procesos *Activos* y el de *Expirados*. El de activos contiene a los procesos que no han consumido todo su *timeslice*, mientras que el de expirados, son todos aquellos que ya lo han agotado.

Cuando una tarea consume todo su *timeslice*, este es recalculado justo antes de moverla al array de expirados. Cuando el array de activos se queda vacío, entonces se intercambian los punteros que hay en la estructura `runqueue_t` (ver sección 2.1.2, página 17). De esta manera, el recorrido del bucle para calcular los nuevos *timeslices* se reduce a intercambiar dos punteros, lo cual es independiente del número de procesos en ejecución.

Estas acciones son llevadas a cabo por los métodos `scheduler_tick()` y `schedule()` (sección 2.2, página 30) respectivamente.

### 2.1.4. Dominios de Planificación

Los Dominios de Planificación (*Scheduling Domain*) son conjuntos de procesadores que comparten ciertas propiedades y una misma política de planificación, de manera que pueden repartirse la carga de trabajo. Estos conjuntos son multinivel, ya que se intenta imitar la jerarquía *hardware* del sistema.

Este mecanismo surge de la necesidad de proporcionar un sistema de planificación flexible y genérico, que permita gestionar la carga de trabajo en las diferentes topologías de los sistemas multiprocesadores actuales. En especial, porque según los distintos niveles de esta jerarquía, los procesadores se relacionan entre sí de manera distinta.

Por ejemplo, en una CPU SMT (*Symmetric Multi-Threading*), los procesadores lógicos comparten la memoria principal, las memorias *caches* e incluso las unidades funcionales. En este caso, no existe *afinidad a la memoria cache* (*cache affinity*), de manera que cualquier proceso que estuvo ejecutándose en uno de estos procesadores y fue suspendido, puede ser reactivado en otro sin perder necesariamente los datos que tenía en esa memoria, y por tanto, sin generar fallos de acceso. El equilibrado de carga puede realizarse con bastante frecuencia.

Por otro lado, en los sistemas SMP (*Symmetric Multi-Processing*) sólo se comparte la memoria principal, ya que cada procesador posee sus propios recursos internos. Aquí sí existe afinidad a la *cache*, aunque dura muy poco. Por tanto, si no se puede despertar a un proceso en la misma CPU en la que se estuvo ejecutando (que sería el caso ideal), entonces este puede ser activado en otro procesador. Además, no es necesario realizar un equilibrado de carga con demasiada frecuencia. Sin embargo, si las CPUs de este sistema SMP, son SMT, tratar a todos los procesadores lógicos por igual no es la mejor solución, ya que si por ejemplo, hubiesen sólo dos tareas en ejecución, lo ideal sería ponerlas en distintas CPUs físicas y no en los procesadores lógicos de la misma CPU.

Finalmente, en los sistemas NUMA (*Non-Uniform Memory Access*), cada nodo puede tener una velocidad de acceso diferente para las distintas áreas de la memoria principal. La afinidad a la *cache* es bastante duradera y por consiguiente, sólo se debería migrar un proceso de un nodo a otro de manera esporádica, ya que es una operación bastante costosa.

Por tanto, el principal problema al que el planificador debe enfrentarse en un sistema con más de un procesador, es el *equilibrado de carga*, ya que no es deseable tener unos procesadores con mucha carga de trabajo mientras que otros están ociosos. Sin embargo, el mover tareas de un procesador a otro es una operación costosa y debe hacerse de forma razonada. El equilibrado de carga se explica con detalle en la sección 2.3 (página 33). En cuanto a la gestión de los sistemas SMT en Linux, se detalla en el capítulo 3 (página 59).

### Estructura de los Dominios de Planificación

Existen dos tipos de estructuras básicas en este mecanismo: la primera es `struct sched_domain`, donde se almacena toda la información necesaria para el equilibrado de carga, y la segunda es `struct sched_group`, la cual

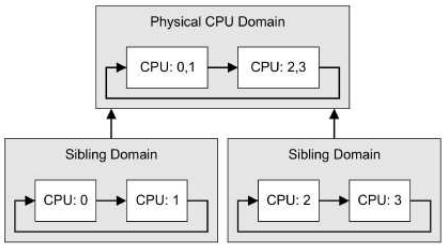


Figura 2.6: *Scheduling Domains* en un sistema SMP con procesadores SMT.

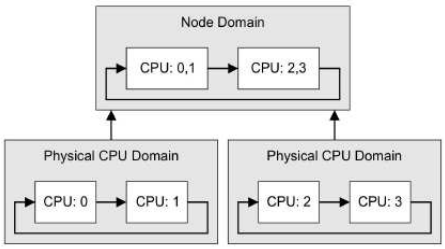


Figura 2.7: *Scheduling Domains* en un sistema NUMA con nodos SMP.

representa a un grupo de procesadores que son tratados como una unidad durante dicho equilibrado. Esto lo podemos ver con los siguientes ejemplos ([23]):

Supongamos el caso de un sistema SMP con dos CPUs SMT y que cada una de ellas, contiene dos procesadores lógicos (o virtuales). Con los Dominios de Planificación, cada CPU física representaría un dominio con una estructura `struct sched_domain`, la cual incluiría sus dos respectivos procesadores lógicos, cada uno en un grupo `struct sched_group`. Estos dos dominios tendrían un dominio padre común, el cual contendría a los cuatro procesadores lógicos agrupados en dos estructuras `struct sched_group`. Esta jerarquía se puede ver en la figura 2.6.

Ahora consideremos un sistema NUMA con dos nodos, cada uno de los cuales es una máquina SMP que contiene dos CPUs no SMT (sin procesadores lógicos). En este caso, cada sistema SMP representa un dominio, que incluye a sus respectivas CPUs en dos grupos. Al igual que en el caso anterior, existe un dominio padre común al de ambos nodos, el cual contiene dos grupos con las cuatro CPUs. Esta nueva jerarquía queda ilustrada en la figura 2.7.

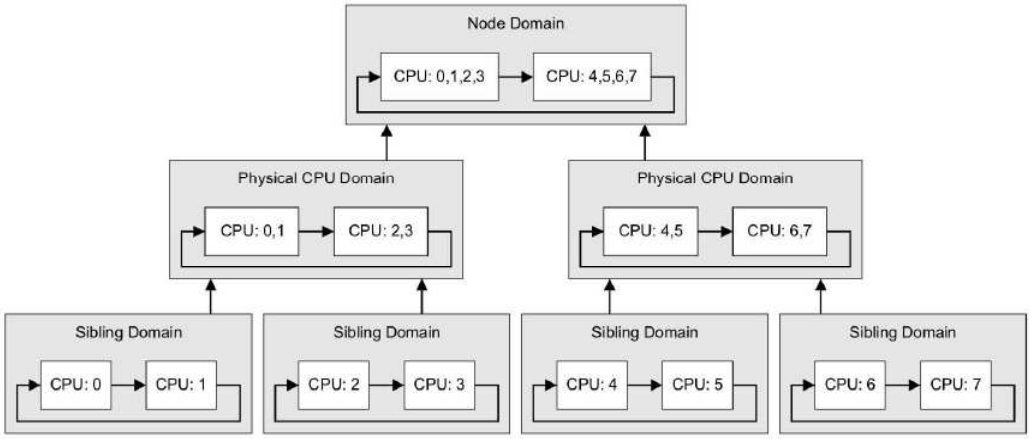


Figura 2.8: *Scheduling Domains* en un sistema NUMA con nodos SMP y procesadores SMT.

Finalmente, un sistema que fuese la combinación de los dos anteriores, presentaría una jerarquía de tres niveles: cuatro dominios SMT agrupados en dos SMP, los cuales tienen un dominio padre común. Esta situación está representada en la figura 2.8.

### Implementación de los Dominios de Planificación

La implementación de la estructura de los dominios de planificación, y sus campos más importantes, son los siguientes:

```
struct sched_domain {
    struct sched_domain *parent;        // Dominio de planificación padre
    struct sched_group *groups;         // Grupos de equilibrado del dominio
    cpumask_t span;                     // Procesadores pertenecientes al dominio
    unsigned long min_interval;         // Mínimo intervalo de equilibrado (ms)
    unsigned long max_interval;         // Máximo intervalo de equilibrado (ms)
    unsigned int busy_factor;           // Factor de reducción del equilibrado si hay mucha carga
    unsigned int imbalance_pct;         // Umbral de desequilibrado
    unsigned long long cache_hot_time;  // Tiempo de validez de los datos en la cache (ns)
    unsigned int cache_nice_tries;      // Número de veces que se dejan los procesos 'cache hot'
    unsigned int per_cpu_gain;          // Ganancia al añadir dominios de CPU
    int flags;                          // Flags de los dominios de planificación.
    unsigned long last_balance;         // Último equilibrado (jiffies)
    unsigned int balance_interval;      // Intervalo entre equilibrados (ms)
    unsigned int nr_balance_failed;     // Número de equilibrados fallidos.
};

struct sched_group {
    struct sched_group *next;           // Puntero a otro grupo de procesadores
    cpumask_t cpumask;                  // Procesadores del grupo
    unsigned long cpu_power;            // Capacidad de computación del grupo.
};
```

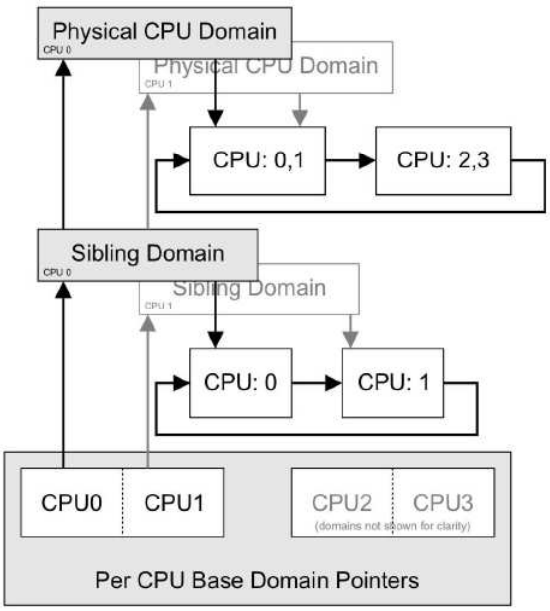


Figura 2.9: Implementación de los *Scheduling Domains* en un sistema SMP con procesadores SMT.

Por razones de eficiencia, la estructura `struct sched_domain` no se comparte entre los procesadores que pertenecen al dominio que esta representa, sino que cada uno tiene su propia copia. Sin embargo, sí que se comparten las estructuras de los grupos de procesadores (`struct sched_group`). Por ejemplo, para el sistema SMP con dos CPUs SMT (cada una con dos procesadores lógicos), explicado anteriormente, el árbol de dominios sería como el que se muestra en la figura 2.9 ([23]).

Al iniciarse el kernel, se establecen unos valores por defecto en los campos de la estructura `struct sched_domain` según la topología del sistema. Estos valores inciales se encuentran definidos en `include/linux/topology.h`, salvo los de los sistemas NUMA, que deben ser definidos por cada arquitectura en `include/asm-*/topology.h`.

### Política de equilibrado de carga

La política de equilibrado de carga se define en cada dominio con una combinación de valores en el campo `flags`. Todos los posibles valores de este campo se pueden ver en la tabla 2.1.

Cuadro 2.1: *Flags* en los *Scheduling Domains*

| *Flags* | *Descripción* |
|---|---|
| `SD_LOAD_BALANCE` | Habilita el equilibrado de carga en el dominio |
| `SD_BALANCE_NEWIDLE` | Hacer equilibrado cuando quede poca carga de trabajo |
| `SD_BALANCE_EXEC` | Realizar equilibrado en las llamadas a `exec()` |
| `SD_WAKE_IDLE` | Despertar los procesos en las CPUs del dominio que estén ociosas |
| `SD_WAKE_AFFINE` | Permite que los procesos sean despertados en otras CPUs del dominio |
| `SD_WAKE_BALANCE` | Equilibrar cuando se despierte un proceso |
| `SD_SHARE_CPUPOWER` | Miembros del dominio comparten los recursos de la CPU |

Tanto en sistemas SMP como SMT, se establecen por defecto los flags `SD_LOAD_BALANCE`, `SD_WAKE_IDLE`, `SD_WAKE_AFFINE` y `SD_BALANCE_NEWIDLE`, ya que las penalizaciones por mover procesos a otras CPUs del mismo dominio, son escasas (o nulas, como en el caso de arquitecturas SMT). Además, también se activa el flag `SD_BALANCE_EXEC`, ya que en una llamada a `exec()`, el proceso pierde toda afinidad con el procesador (ya no necesita los datos que haya en su memoria *cache*), y por tanto es el mejor candidato para ser cambiado de CPU durante un equilibrado de carga.

Sin embargo, el flag `SD_SHARE_CPUPOWER` sólo se establece en arquitecturas SMT para indicar que todos los procesadores del mismo dominio comparten los recursos de la CPU física, y por tanto, hay que dejarlos libres para las tareas más prioritarias. Esto se explica con detalle en el capítulo 3 (página 59).

Del mismo modo, el flag `SD_WAKE_BALANCE` solamente se activa en sistemas SMP para permitir que un proceso sea despertado en otra CPU, del mismo dominio, si existe un desequilibrio en la carga de trabajo. Esto es así, porque en los sistemas SMP, los equilibrados de carga son menos frecuentes que en los sistemas SMT.

Finalmene, en los sistemas NUMA, las arquitecturas que definen valores iniciales no suelen activar los flags de equilibrado de carga en el nivel más alto de la jerarquía, ya que mover procesos de un nodo a otro, es una operación costosa y que ocurre muy esporádicamente.

## 2.2. La función de planificación

La función principal del planificador es `schedule()`. Su propósito es seleccionar el siguiente proceso a ejecutar. Esta función es invocada cuando un proceso desea ceder el procesador (a través del método `sched_yield()`[2]), o cuando este debe ser expropiado (ver sección 2.4.3, página 43).

Podemos dividir los pasos llevados a cabo por este algoritmo en cuatro secciones o partes:

**Preparativos** : lo primero es realizar una serie de preparativos y comprobaciones, tales como:

- Verificar que la función no fue invocada mientras el *kernel* está en modo atómico o con las interrupciones deshabilitadas, ya que podría provocar interbloqueos.
- Verificar que el proceso en ejecución no es la tarea "ociosa" (aquella que se ejecuta cuando no hay otros procesos a ejecutar) y que esta no está en estado `TASK_RUNNING`.
- Deshabilitar la expropiación y determinar el tiempo que el proceso actual ha estado ejecutándose. Si el proceso era interactivo, dicho tiempo es reducido para evitar que este tipo de procesos, que suelen estar esperando en operaciones de E/S, pierdan su estatus de "interactivo" debido a que puntualmente han estado ejecutándose durante un largo período de tiempo.
- Si se está en una expropiación, las tareas en estado `TASK_INTERRUPTIBLE` y con una señal de interrupción pendiente pasan a estado `TASK_RUNNING`, mientras que las que están en estado `TASK_UNINTERRUPTIBLE` se eliminan de la cola de tareas. Esto es así, porque los procesos "interrumpibles" y con una señal pendiente, necesitan tratarla, mientras que los "ininterrumpibles" no deberían estar en la cola de procesos.

[2] ver sección 2.5.3, página 53

**Buscar procesos candidatos** : ahora se verifica si hay tareas a ejecutar. Esta acción se realiza de la siguiente manera:

- Si no hay procesos para ejecutar en la cola, se procede a un equilibrado de carga (ver sección 2.3, página 33). Si aun así, no hay tareas, entonces se elige a la tarea ociosa (*idle task*). Además, se invoca a la función `wake_sleeping_dependent()`, que pertenece a la *Planificación SMT*[3].
- Si por el contrario, había tareas para ejecutar, primero se verifica si se puede seleccionar una de ellas o deben dejarse suspendidas. Esto se realiza mediante la función `dependent_sleeper()`, la cual también pertenece a la Planificación SMT, y que por tanto, solo tendrá efectos si está instalado dicho tipo de planificación.
- Si en este momento, ya no quedan procesos en el Array de tareas activas, se hace un intercambio entre los dos arrays, lo cual equivale a haber hecho el bucle de recálculo de *timeslices* (ver sección 2.1.3, página 23).

**Seleccionar tarea a ejecutar** : en este momento ya hay tareas a ejecutar en el array de procesos activos. Para seleccionar la siguiente tarea a ejecutar se realizan los siguiente pasos:

- Se invoca a la función `sched_find_first_bit()` para que localice de manera eficiente, en el campo `bitmap[]` del array, el nivel de prioridad más alto donde haya alguna tarea ejecutable. Con esto, se evita tener que recorrer el array buscando dicha tarea.
- Selecciona en el array, la primera tarea de la lista que corresponde al nivel de prioridad encontrado. Por ejemplo, si la función `sched_find_first_bit()` devuelve 7, entonces el proceso de mayor prioridad es el primero de la lista de tareas que se encuentra en `array->queue[7]`. Esto se puede ver mejor con la figura 2.10 ([20]) y en la descripción de los campos de los *Arrays de prioridad* (sección 2.1.3, página 21).
- Si el proceso seleccionado no es de Tiempo Real, estaba suspendido y en un estado distinto a `TASK_UNINTERRUPTIBLE`, es decir, el valor del campo `activated` de su `task_t` es mayor que 0, entonces es muy probable que se trate de un proceso interactivo, y por tanto, se invoca a la función `recalc_task_prio()` para que actualice

[3]Todo lo referente a la planificación SMT, se encuentra explicado en el capítulo 3, página 59.

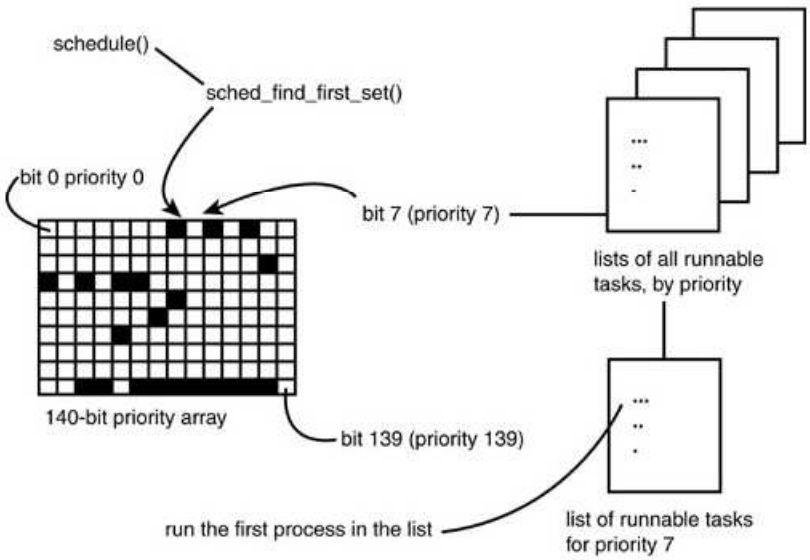


Figura 2.10: Algoritmo de planificación en Linux 2.6.X.

su `sleep_avg` (el tiempo medio que pasa durmiendo) y modifique su prioridad. Finalmente se actualiza su posición en el Array de prioridad. Todo lo referente al campo `sleep_avg` y las prioridades, se explica con detalle en la sección 2.4.1 de la página 38. En cuanto al campo `activated`, ver sección 2.5.2, página 50.

**Cambio de contexto** : una vez seleccionado el siguiente proceso a ejecutar, se procede al cambio de contexto entre este y el proceso anterior de la siguiente manera:

- Desactiva el flag `TIF_NEED_RESCHED` en el proceso que va a ser expropiado (ver sección 2.4.3, página 43).
- Actualiza el tiempo que el proceso ha estado ejecutándose desde el último cambio de contexto o *scheduler tick* (interrupción del reloj del sistema).
- Actualiza las marcas de tiempo del proceso.
- Realiza el cambio de contexto (explicado en detalle en la sección 2.5.1, página 49).
- Habilita la expropiación la cual había sido desactivada durante la planificación. Si durante este tiempo, alguna expropiación había sido solicitada, el algoritmo vuelve a comenzar.

## 2.3. Equilibrado de carga

Como Linux implementa colas de procesos separadas para cada procesador de forma que, a todos los efectos, se realiza la planificación para cada procesador de forma individual. Para poder llevar una política de planificación a nivel global, Linux implementa los equilibradores de carga.

Durante la creación y eliminación de nuevos procesos, o debido a procesos que se envían a colas de espera (ver 2.5.2), se pueden producir lo que se llama desequilibrios en la carga. Si lanzas varios trabajos, la mitad a cada CPU, y mueren o son suspendidos todos los procesos que tienes en una de las CPUs, te encuentras con una CPU llena de procesos compitiendo por su uso y otra CPU totalmente ociosa. El equilibrado de carga es el mecanismo para evitar este tipo de situaciones.

Un equilibrador de carga realiza el equilibrado comparando la carga de la cola de ejecución de su procesador con la de los otros. Si detectase un desequilibrio entre ellas movería tareas desde la cola más ocupada a la más ociosa.

### 2.3.1. Equilibrado activo y pasivo

Existen dos formas de equilibrado de carga: el equilibrado de carga activo y el equilibrado de carga pasivo. El equilibrado de carga pasivo es el que se realiza cuando se produce algún evento que provoque una posible situación de desequilibrio. Por otro lado, el equilibrado de carga activo se asegura, cada cierto tiempo, de que las colas siguen equilibradas.

El objetivo que persiguen los equilibradores es que haya, en la medida de lo posible, el mismo número de procesos corriendo en cada CPU, aunque se podría programar una política de planificación global distinta reprogramando los equilibradores.

#### Equilibrado pasivo

El equilibrado de carga pasivo se realiza ante los siguientes eventos:

- Una cola de ejecución se va a quedar vacía: cuando el planificador, al sacar de la cola de ejecución un trabajo, se encuentra con que no quedan trabajos en dicha cola, llama a la función `void idle_balance(int this_cpu, runqueue_t *this_rq)` con la

CPU que se va a quedar ociosa y su correspondiente cola de ejecución como parámetros.

`void idle_balance(int this_cpu, runqueue_t *this_rq)` recorre los dominios de planificación a los que pertenece la CPU y, si tienen activo el flag `SD_BALANCE_NEWIDLE`, llama a `int load_balance_newidle(int this_cpu, runqueue_t *this_rq, struct sched_domain *sd)` para que busque en ese dominio el grupo más ocupado, y de ese grupo, la cola con más tareas. La función se apoya en `int move_tasks(...)` para que migre todas las tareas que pueda de esa cola hacia la CPU que se va a quedar ociosa, hasta que ambas colas queden equilibradas.

Para saber si una tarea es candidata a ser migrada, utiliza la función `int can_migrate_task(...)` que rechaza para la migración las tareas que:

1. Están corriendo.
2. No son afines a la CPU de destino.
3. Son *cache-hot* en su CPU, es decir, las que es muy probable que estén total o parcialmente en cache (porque han estado corriendo recientemente, por ejemplo).

Con esto se evita una migración que resultase en pérdidas grandes de rendimiento.

- Un proceso ha realizado un `execve()`: si un proceso ha realizado una llamada a `execve()` pierde toda la afinidad con la cache y la memoria, así que es un candidato perfecto para una migración. Desde la función `void sched_exec()` (que es invocada desde la llamada al sistema que realiza el `execve()`), se busca la CPU más ociosa del dominio que tenga el flag `SD_BALANCE_EXEC` activo (ver 2.1.4) y se migra la tarea que realiza el `execve` a esa CPU. La migración en este caso la realizan los hilos de migración (2.3.2).

- Un proceso ha realizado un `fork()`: si un proceso hace una llamada a `fork()`, podría haberse producido un desequilibrio con la creación de la nueva tarea, de modo que se intenta un reequilibrado. La forma en la que se produce es la misma en la que se produce el equilibrado activo (2.3.1), desde `rebalance_tick()`. Cuando al proceso que hace

el `fork()` llama a `scheduler_tick()` para preparar la planificación del proceso recién creado, y desde esa función se hace una llamada a `rebalance_tick()`.

### Equilibrado activo

El equilibrado de carga activo entra en juego cada cierto tiempo para asegurarse de que la carga dentro de un dominio sigue equilibrada. El periodo de tiempo entre llamadas depende de la carga de la CPU, y cuanto más homogénea tienda a ser la carga, más largo es el periodo de equilibrado activo.

Los periodos de equilibrado activo, que evidentemente están asociados a cada dominio, se guardan en el campo `balance_interval` de la estructura `sched_domain` (ver 2.1.4). Dichos periodos se miden en milisegundos. Además, el periodo de equilibrado se desplaza para cada CPU ($HZ * cpu/NR\_CPUs$), de forma que evitamos que el equilibrado activo se realice en todas las CPUs al mismo tiempo. Para saber si toca reequilibrar la carga también guarda el tiempo de su último equilibrado en el campo `last_balance` del del `sched_domain`.

La función de equilibrado activo es `rebalance_tick()` y se invoca desde `scheduler_tick()` como se ha comentado más arriba. Esta función lo primero que hace es recalcular la carga de la CPU en la que ha sido invocada, y para cada dominio de planificación al que pertenece la CPU y tiene activado el equilibrado de carga (con el flag `SD_LOAD_BALANCE`), hace lo siguiente:

1. Obtener el intervalo de planificación del dominio.

2. Si la CPU no estaba ociosa, modifica ese intervalo multiplicándolo por un factor (`busy_factor`) asociado al dominio, de esta forma si la CPU está ocupada los periodos de reequilibrado son más largos.

3. Transforma el periodo de equilibrado de milisegundos a *jiffies*, que en nuestro caso son milisegundos también (arquitecturas *i386* desde *Linux 2.5.x*).

4. Si el tiempo transcurrido desde el último equilibrado es mayor que el intervalo de equilibrado, se llama a `load_balance()`.

5. Actualizamos el tiempo del último equilibrado.

La función `load_balance()` comprueba que la CPU en la que se invoca está equilibrada dentro de su dominio e intenta mover tareas a la CPU en caso de que no lo estuviese. Para ello busca el grupo más ocupado de entre los desequilibrados y dentro de él la cola más ocupada. Si no hay un grupo o una cola lo suficientemente ocupada, duplica el periodo de equilibrado (`balance_interval`). De esta forma, cuanto más tiempo permanezca un dominio equilibrado, menos frecuentes serán los equilibrados. Así es como se consigue optimizar el caso de una carga homogénea en el dominio.

En caso de encontrar una cola de ejecución desequilibrada, las colas se bloquean y se llama a `move_tasks()` (página 33) para que mueva, de las tareas que le han pedido, todas las posibles. Si `move_task()` no pudiese mover ninguna tarea, después de un cierto número de llamadas a `load_balance()` se intentaría un equilibrado apoyándose en los hilos de migración (2.3.2) para hacer equilibrado activo.

### 2.3.2. Hilos de migración

Los hilos de migración son *kernel threads* de tiempo real, uno asociado a cada CPU, que se encargan del trabajo de migrar tareas de su CPU a las demás. Se inicializan en `migration_call()`, función que es llamada tras inicializarse cada CPU, y están esperando tareas para migrar hasta que la CPU se da de baja. Además de para migrar tareas, se pueden utilizar para establecer un dominio de planificación a una determinada cola de forma segura.

Los hilos de migración, utilizados para migrar tareas, pueden funcionar de dos maneras: se les puede pasar una lista de tareas que deseamos migrar, o se les puede pedir que hagan un equilibrado activo cuando se desea expropiar la CPU a una tarea que nos impide la migración.

El segundo caso es el explicado en la parte de equilibrado de carga activo (2.3.1), el hilo de migración se limita a llamar a la función `active_load_balance()` que busca desequilibrios hacia otras colas de ejecución en el dominio y mueve tareas hacia ellas. El caso de la lista de tareas a migrar es un poco más complejo y pasamos a describirlo con más detalle a continuación.

Para pedir a un hilo que realice migraciones de tareas determinadas hay que añadir peticiones a la cola de migración de su cola de ejecución. La cola

de migración es una lista enlazada, el campo `migration_queue` de las colas de ejecución, en la que el hilo de migración espera encontrar estructuras del tipo `migration_req_t` especificando la migración.

Dichas estructuras contienen los siguientes campos:

- `list`: la cola de migración a la que se ha añadido.
- `request_type`: especifica el tipo de petición que se le hace al hilo de migración. Como ya se ha explicado puede ser de dos tipos:
  1. `REQ_MOVE_TASK`: para mover una tarea de su cola a otra.
  2. `REQ_SET_DOMAIN`: para cambiar el dominio de una cola de ejecución.
- `task`: puntero a la tarea que se quiere mover.
- `dest_cpu`: CPU a la que se desea mover la tarea.
- `sd`: dominio al que se quiere cambiar.
- `done`: variable de condición. Se utiliza para poder esperar a que el hilo de migración termine su trabajo. Al finalizar su trabajo el hilo hace un `complete()` sobre la variable de forma que todos los procesos que estén esperando con un `wait_for_completion()` se desbloquean.

El hilo de migración utiliza la función `__migrate_task()` para mover la tarea de una CPU a otra. Esta función bloquea las dos colas implicadas, saca la tarea a migrar de su cola, la sincroniza con la cola de destino y la añade. Si esa tarea expropia por prioridad a la que esté corriendo en la cola de destino, se hace una replanificación para ajustarse a las prioridades de las tareas.

## 2.4. Política de planificación

La Política de planificación es la responsable de un aprovechamiento óptimo del procesador, ya que es la que determina qué proceso debe ejecutarse y cuándo. Además debe hacerlo satisfaciendo dos objetivos contrarios: *baja latencia* (tiempo de respuesta rápido) y un *alto rendimiento* (*throughput*).

Para cumplir estos objetivos se diferencia entre dos tipos de procesos: los *I/O-bound* y los *CPU-bound*. Los primeros son aquellos que pasan la mayor parte del tiempo suspendidos esperando a que finalicen operaciones de E/S, mientras que los segundos están continuamente utilizando los recursos del procesador.

Los procesos *I/O-bound* solo se pueden ejecutar durante periodos de tiempo cortos, ya que terminarán por suspenderse en alguna operación de E/S. Por el contrario, los *CPU-bound* pueden estar en ejecución hasta que sean expropiados.

Por tanto, para favorecer la baja latencia, se otorga mayor prioridad a los procesos *interactivos* (*I/O-bound*) para que estos puedan ejecutarse un mayor número de veces. De esta manera, pueden comenzar cuanto antes sus peticiones de E/S y dejar al procesador "libre" para los procesos que más lo necesitan: los *CPU-bound*.

El cálculo de prioridades y de tiempos de ejecución (*timeslices*) se explica con detalle en las siguientes subsecciones.

### 2.4.1. Prioridades de los procesos

Linux emplea una planificación *basada en prioridades* la cual consiste en clasificar a todos los procesos según su valor y necesidades de tiempo en el procesador. De esta manera, los procesos con mayor prioridad siempre se ejecutarán antes que los de menor prioridad. Si existen varios con una misma prioridad, estos se irán seleccionando por turnos rotatorios (*round-robin*).

Las prioridades en Linux se encuentran representadas internamente por valores pertenecientes al rango [0..`MAX_PRIO-1`]. Por defecto la constante `MAX_PRIO` (definida en `include/linux/sched.h`) vale 140 y este es por tanto, el número de niveles de prioridad que existen en el sistema. Cuanto menor sea este nivel para un proceso, *mayor será su prioridad*.

Estos niveles se encuentran divididos en dos subrangos. El primero de ellos es para procesos de *Tiempo Real* y comprende a los primeros `MAX_RT_PRIO` niveles, es decir, el rango [0..`MAX_RT_PRIO-1`], donde la constante `MAX_RT_PRIO` (también definida en `include/linux/sched.h`) vale por defecto 100. Todo lo referente a los procesos de Tiempo Real se explica en la sección 2.4.4, página 47.

El segundo subrango ([`MAX_RT_PRIO`..`MAX_PRIO-1`]) que es el que se tratará en esta subsección, incluye a los 40 niveles restantes. En estos niveles estarán la mayoría de los procesos de usuario.

### Prioridades Estáticas

Cuando se crea un proceso de usuario, se le asocia una prioridad base igual a la del proceso padre. Esta prioridad es denominada *Prioridad Estática*, ya que es la que tendrá este nuevo proceso a lo largo de su vida. El nivel de prioridad al que pertenecen estos procesos es almacenado en el campo `static_prio` de la estructura `task_t` (ver sección 2.1.1, página 12).

Es posible modificar la prioridad estática de un proceso de usuario mediante la llamada al sistema `nice()`. Al invocarla, se ejecuta en `kernel/sched.c` el método `sys_nice()`, el cual, aparte de algunas comprobaciones, llama a `set_user_nice()`. Esta función establece la nueva Prioridad Estática del proceso y actualiza su posición en el Array de Prioridades.

Sin embargo, en esta llamada al sistema, el valor de la nueva prioridad introducido como parámetro, no es uno de los 40 niveles de prioridad en el que se ejecutan los procesos de usuario, sino una conversión al rango [-19..0..20]. Los valores de este nuevo rango se denominan también *nice*. De esta manera, los procesos con mayor prioridad serán aquellos que tengan un *nice* negativo, mientras que los positivos tendrán menor prioridad (están siendo "amables" con el resto de procesos al reducir su prioridad).

Se puede consultar el *nice* de un proceso de usuario, a través de las funciones `task_nice()` y `task_prio()` en `kernel/sched.c`

### Prioridades Dinámicas y Prioridad Efectiva

Tal y como se mencionó al inicio de esta sección, el planificador de Linux siempre favorece a los procesos interactivos y penaliza a los intensivos en cálculo. Para llevar a cabo esta acción, modifica ligeramente las prioridades estáticas de los procesos, aumentando la de los primeros (disminuyendo su nivel de prioridad) y reduciendo la de los segundos (incrementando su nivel). En concreto, este ajuste en sus prioridades consiste en "bonificar" o penalizar a los procesos con hasta 5 niveles de prioridad. A estas bonificaciones y penalizaciones se les denomina *Prioridades Dinámicas*, ya que el ajuste se realiza de manera continua y siguiendo el comportamiento de cada proceso.

El nuevo valor de la prioridad queda almacenado en el campo `prio` de la estructura `task_t` de cada proceso, ya que este es el campo que realmente indica la prioridad de todos los procesos (ya sean de usuario o de tiempo real) y el utilizado como índice a la hora de insertarlo en los Arrays de Prioridad.

Para saber qué procesos deben ser penalizados o beneficiados en sus prioridades, el planificador emplea una heurística basada en *el tiempo medio que cada proceso pasa suspendido* (muy probablemente en operaciones de E/S). Así, cuanto mayor sea la media de un proceso, mayor será la reducción en sus niveles de prioridad (hasta un límite de 5 niveles por debajo de su prioridad estática). Este tiempo medio se guarda en cada proceso en el campo `sleep_avg` de la estructura `task_t`.

Cuando un proceso se despierta, el tiempo que ha pasado suspendido pasa a formar parte de su media. Esta acción es realizada en la función `recalc_task_prio()`, la cual es invocada desde `activate_task()`. También es invocada desde `schedule()` cuando el proceso seleccionado para ejecución estaba suspendido, ya que además se contabiliza el tiempo que pasa en la *runqueue* desde que es despertado hasta esa primera selección.

Sin embargo, esta función no incrementa el valor de la media si el proceso estaba suspendido en estado `TASK_UNINTERRUPTIBLE` (el campo `activated` de su estructura `task_t` vale -1), ya que podría tratarse de un *CPU-bound* que ocasionalmente estaba suspendido en una operación de E/S. Finalmente, invoca a `effective_prio()` para que actualice su prioridad y la almacene en el campo `prio` de su estructura `task_t`.

La función `effective_prio()` es la calcula el "bonus" o la penalización en las prioridades. Además de ser invocada desde `recalc_task_prio()`, también lo es desde `scheduler_tick()`. En cualquier caso, es siempre después de haber modificado el `sleep_avg` de un proceso, ya que esto es la base de la heurística. Básicamente lo que realiza es escalar el tiempo medio que un proceso pasa suspendido ([0..`MAX_SLEEP_AVG`]) al rango de bonus/penalizaciones [-5..0..+5]. La constante `MAX_SLEEP_AVG` (definida en `kernel/sched.c`), cuyo valor por defecto es 10ms, indica el valor máximo que puede alcanzar `sleep_avg`.

A pesar de esta modificación en la prioridad, como sólo afecta al 25 % del los 40 niveles de prioridad pertenecientes a los procesos que no son de Tiempo Real, se consigue "respetar" la elección del usuario del *nice* de los

procesos. Por ejemplo, un proceso interactivo con *nice* +19 (nivel de prioridad 139) nunca podrá expropiar a un proceso intensivo en cálculo que tenga un *nice* 0 (nivel 120), o bien, un proceso *CPU-bound* de *nice* -20 (nivel 100), nunca será expropiado por uno interactivo de *nice* 0.

#### Equidad al crear y destruir procesos

Un proceso recién creado siempre recibe como *nice* inicial, la prioridad estática de su proceso padre. Además, la función `wake_up_new_task()` reduce el `sleep_avg` tanto del proceso padre como del hijo recién creado. Esto impide que un proceso interactivo que tenga una media alta genere muchos procesos hijos, también con una media alta, y terminen monopolizando al procesador.

Por otro lado, cuando una tarea muere, si su media es menor que la de su proceso padre, entonces también se reduce la media de este último. Esto se realiza en la función `sched_exit()`.

### 2.4.2. *Timeslices*

El *timeslice* de un proceso es el tiempo que este puede estar ejecutándose en el procesador antes de ser expropiado. En general, y para cualquier Sistema Operativo, el determinar un valor por defecto no es una tarea sencilla, ya que uno muy largo reduce la interactividad del sistema e incluso produce la inanición del resto de procesos, mientras que uno muy corto reduce significativamente el rendimiento debido a la sobrecarga del planificador y los cambios de contexto. Por otro lado, los procesos *CPU-bound* necesitan *timeslices* largos (por ejemplo, para mantener sus datos en la cache), mientras que a los *I/O-bound* les basta con uno corto.

Para que el planificador del Linux favorezca a los procesos interactivos, sin penalizar excesivamente a los intensivos en cálculo, se ha optado por darles más prioridad a los primeros, pero otorgando a todos un *timeslice* por defecto (es decir, para el nivel de prioridad 120) relativamente alto (100ms). En cualquier caso, el *timeslice* de un proceso es calculado únicamente en función de su *Prioridad Estática* mediante la función `task_timeslice()`.

#### Cálculo del *timeslice*

La función `task_timeslice()` convierte el rango de las prioridades de los procesos de usuario ([`MAX_RT_PRIO`..`MAX_PRIO-1`]), al rango de tiempos

Cuadro 2.2: *Timeslices* del Planificador

| *Timeslice* | *Cantidad* | *Prioridad estática* | *Nice* |
|---|---|---|---|
| Inicial | La mitad de el del padre | La del padre | El del padre |
| Mínimo (MIN_TIMESLICE) | 5ms | Baja | Positivo |
| Por defecto | 100ms | Normal | 0 |
| Máximo | 800ms | Alta | negativo |

[800ms..100ms..5ms] basándose en el campo `static_prio` (prioridad estática) de los procesos. Cuanto menor sea este valor para un proceso, mayor será su tiempo de ejecución. Sin embargo, cualquier proceso, por muy poca prioridad (nivel alto) que tenga, recibirá un valor superior o igual a `MIN_TIMESLICE` (5ms).

Existe una llamada al sistema para obtener el *timeslice* de un proceso. Se trata de `sched_rr_get_interval()` (método `sys_sched_rr_get_interval()`, definido en `kernel/sched.c`), aunque aparte de algunas comprobaciones, es una simple llamada a `task_timeslice()`.

Podemos resumir los *timeslices* que reciben los procesos según su prioridad estática en la tabla 2.2.

### Actualizando el *timeslice* de un proceso

El *timeslice* de los procesos se guarda en el campo `time_slice` de la estructura `task_t` y se actualiza en la función `scheduler_tick()`, la cual es llamada regularmente a través de una interrupción del reloj.

Normalmente, cuando un proceso consume completamente su tiempo de ejecución se le otorga un nuevo *timeslice*, se calcula su prioridad con la función `effective_prio()` y se mueve del array de procesos activos al de expirados. Además se invoca a `set_tsk_need_resched()` para que el planificador lo expropie y seleccione a otra tarea para su ejecución.

Sin embargo, para impedir que los procesos interactivos se queden mucho tiempo en el Array de expirados, esperando a que todos los demás procesos de menor prioridad se suspendan o consuman sus respectivos *timeslices* (sobre todo los *CPU-bound*, que se ejecutarán hasta que sean expropiados), estos primeros son reinsertados en el Array de Activos (al final de la lista

de procesos de su nivel de prioridad). Esto no significa que vuelven a entrar en ejecución inmediatamente, sino que en un futuro serán seleccionados por el planificador. La interactividad del proceso es determinada con la macro `TASK_INTERACTIVE()`.

No obstante, para prevenir la inanición de los procesos en el array de expirados, se consulta a la macro `EXPIRED_STARVING()` para saber si ha transcurrido "mucho tiempo" desde el último intercambio de arrays (ver sección 2.1.3, página 23), en cuyo caso el proceso interactivo no es reinsertado en el Array de activos sino en el de expirados.

Finalmente, también se evita que los procesos interactivos con un *timeslice* muy alto puedan consumirlo completamente en un solo "turno de ejecución", monopolizando así el procesador. En estos casos se expropian y también son reinsertados en el Array de activos, al final de la lista de procesos de su nivel de prioridad. Su *timeslice* no se pierde sino que es como si se dividiese en trozos más pequeños. Esto se controla con la macro `TIMESLICE_GRANULARITY()`.

#### Equidad al crear y destruir a procesos

Al crear un proceso, el *timeslice* sin consumir del proceso padre se divide a partes iguales entre este y el hijo, de manera que el tiempo de ejecución total no cambia. Esto se realiza en la función `sched_fork()`.

De la misma manera, cuando un proceso hijo muere, su *timeslice* sobrante es retribuido al padre, recuperando así, parte del tiempo de ejecución que perdió al crearlo. Esta acción se lleva a cabo en el método `sched_exit()`.

### 2.4.3. Expropiación

Una de las nuevas características del Linux 2.6 es que es *completamente expropiativo*, es decir, puede expropiar a un proceso independientemente del modo en el que se encuentre el S.O. (usuario o núcleo). Esto ha obligado a cambiar la forma de gestionar los recursos del *kernel*, en especial las estructuras de datos las cuales fueron dotadas de mecanismos de bloqueo (*locks*) para controlar los accesos concurrentes por parte de varios procesos (ver, por ejemplo, las *runqueues* en la sección 2.1.2, página 17).

### Sistemas Operativos Cooperativos y Expropiativos

Para explicar la expropiación, debemos introducir primero dos conceptos: los Sistemas Operativos *Cooperativos* y los *Expropiativos*.

En los primeros, cualquier proceso puede ejecutarse hasta que voluntariamente cede el procesador. Esta aproximación tiene como inconveniente el que un proceso podría monopolizar el sistema por mucho más tiempo que lo que el usuario desea. Peor aún, si se bloquea jamás cederá el procesador y también bloqueará al resto del sistema.

Por el contrario, en los Sistemas Operativos Expropiativos es el Planificador el que decide el tiempo durante el cual cada proceso podrá estar ejecutándose. Dicho tiempo es denominado *timeslice* y cuando una tarea lo consume completamente, el planificador selecciona a otra para su ejecución. También debe hacerlo si existe otro proceso que tenga mayor prioridad, es decir, debe *interrumpir* a la tarea actual para seleccionar la de mayor prioridad.

### Cómo y cuándo invocar al planificador

El *kernel* debe saber cuándo se debe llamar a la función `schedule()` sin esperar a que un proceso lo haga explícitamente, ya que entonces este último podría ejecutarse indefinidamente. Para ello, existe un *flag* denominado `TIF_NEED_RESCHED`, el cual está definido en `include/asm-*/thread_info.h` y se almacena en la estructura `thread_info` de cada proceso. Dicho flag es consultado en distintas ocasiones por el núcleo, de manera que cuando este último detecta a un proceso que lo tiene activado, invoca a `schedule()` para que seleccione a una nueva tarea (los instantes en que se consulta al flag y se procede a la expropiación, se explican con detalle más adelante, en siguientes subsecciones).

El flag es activado en el método `scheduler_tick()` cuando un proceso consume todo su *timeslice*, en `try_to_wake_up()` cuando se despierta a un proceso con mayor prioridad que el que está en ejecución, en `wake_up_new_task()` por la misma razón pero al crear un nuevo proceso, y también durante el equilibrado de carga (ya que un proceso que es migrado a otro procesador puede tener mayor prioridad que el que se está ejecutando en ese procesador destino –Ver sección 2.3, página 33). En cualquier caso, es el planificador (en el método `schedule()`) el que lo desactiva cuando ya ha seleccionado una nueva tarea a ejecutar.

Existen una serie de métodos definidos en `include/linux/sched.h` para acceder a este flag. Dichos métodos son:

- `set_tsk_need_resched(p)`: activa el flag en el proceso `p`.
- `clear_tsk_need_resched(p)`: desactiva el flag en el proceso `p`.
- `need_resched()`: consulta el valor de `TIF_NEED_RESCHED` en el proceso que está en ejecución.

Sin embargo, cuando se compila el núcleo con la opción `CONFIG_SMP`, se debe emplear el método `resched_task(p)` ya que el proceso `p` podría estar asignado a otra CPU distinta y por tanto hay que invocar al Planificador *en ese procesador*.

El flag existe en cada proceso porque es más eficiente acceder a un valor que se encuentra en un Descriptor de Proceso que a uno que esté en una variable global, especialmente si se trata del proceso `current`, ya que su descriptor estará en la cache del procesador. En las versiones anteriores al Linux 2.2 era una variable global. Luego, en el 2.2 y el 2.4, era un campo de tipo `int` en la estructura `task_struct`. Finalmente, en esta versión 2.6 se ha convertido en un simple bit almacenado en la estructura `thread_info` la cual es uno de los campos de la `task_struct` (o `task_t`), tal y como se mencionó anteriormente.

Sin embargo, no siempre es "seguro" hacer una expropiación. Tal y como veremos en las siguientes subsecciones, el *kernel* expropia a los procesos sólo en determinadas situaciones y cuando se cumplen ciertas condiciones.

### Expropiación de usuario

La expropiación de usuario ocurre cuando el flag está activado y el kernel está a punto de volver al modo usuario, bien sea porque ha acabado una llamada al sistema o porque está volviendo de una rutina de tratamiento de interrupción. En este momento el kernel consulta al flag puesto que sabe que es seguro continuar ejecutando el mismo proceso o bien seleccionar a uno nuevo. El código que se ejecuta al volver de una llamada al sistema, o de una rutina de tratamiento de interrupción, depende de la arquitectura del procesador y está implementado en ensamblador en `arch/*/kernel/entry.S`.

En resumen, el flag se consulta cuando:

- Se vuelve al modo usuario desde una llamada al sistema.
- Se vuelve al modo usuario desde una rutina de tratamiento de interrupción.

### Expropiación de Núcleo

En versiones anteriores de Linux, la expropiación sólo se podía llevar a cabo cuando el proceso que estaba en ejecución estaba en modo usuario, es decir, no se podía expropiar si estaba en modo kernel. Sin embargo, en la versión 2.6 es posible expropiar a un proceso en cualquier momento (incluso si se está ejecutando código del propio *kernel*), siempre y cuando sea seguro de hacerlo. Para ello, el núcleo debe estar compilado con la opción `CONFIG_PREEMPT`.

Es seguro expropiar a un proceso cuando este no mantiene ningún bloqueo sobre alguna estructura de datos o zona de código, ya que hacerlo en caso contrario impediría el acceso a esos recursos por parte de otras tareas. Para saber cuando se puede expropiar, cada proceso cuenta con un contador denominado `preempt_count` que está almacenado en su estructura `thread_info`. El contador comienza valiendo cero, se incrementa cada vez que el proceso adquiere un *lock* y se decrementa cuando lo libera. Cuando el contador vale cero, entonces es seguro expropiarlo. Se puede acceder al contador del proceso en ejecución (`current`) con las macros `inc_preempt_count()`, `preempt_count()` y `dec_preempt_count()` definidas en `include/linux/preempt.h`.

Cuando se vuelve de una interrupción y se regresa al modo núcleo, el kernel consulta los valores del contador `preempt_count` y el flag `TIF_NEED_RESCHED`:

- Si `TIF_NEED_RESCHED` está activado y `preempt_count` vale cero: existe un proceso de mayor prioridad que el actual y por tanto, es seguro (y se debe) expropiarlo.
- Si `preempt_count` es mayor que cero: el proceso actual posee algún *lock* y no es seguro expropiarlo. En este caso se continúa con la ejecución de la tarea y cuando el proceso libere todos los bloqueos que tenga (`preempt_count` volverá a valer cero), el código que hizo los desbloqueos (por ejemplo, la función `spin_unlock()`) chequeará el flag `TIF_NEED_RESCHED`. Si este está activado, entonces se invocará al planificador (método `schedule()`).

#### Desactivar la expropiación

Es posible desactivar y reactivar la expropiación mediante las macros `preempt_disable()` y `preempt_enable()`, las cuales simplemente modifican el valor de `preempt_count`. Además, `preempt_enable()` consulta el flag `TIF_NEED_RESCHED` e invoca al planificador en caso de estar activado.

Estas macros son utilizadas en las funciones de bloqueo `spin_lock()`, `spin_unlock()` las cuales son empleadas para proteger secciones críticas en el código. De hecho, son a su vez invocadas por las funciones de bloqueo de las *runqueues* (ver sección 2.1.2, página 17).

Para más detalles sobre cómo proteger el código de la expropiación, se puede consultar el fichero `Documentation/preempt-locking.txt` en la documentación del código fuente del kernel.

### 2.4.4. Planificación en Tiempo Real

El planificador de Linux también proporciona una planificación en tiempo real suave (*soft real-time*), es decir, una planificación en la que se *intenta* cumplir los plazos de ejecución de los procesos (*deadline*) pero sin tener una garantía de ello.

Tal y como se mencionó al explicar los Niveles de Prioridad en Linux (sección 2.4.1, página 38), el rango de prioridades de los procesos de tiempo real es el comprendido entre 0 y `MAX_RT_PRIO-1`, y que por defecto son los primeros cien niveles. Por tanto, este tipo de procesos siempre expropiarán a los de usuario.

#### Esquemas de planificación

Linux emplea diferentes esquemas o políticas de planificación dependiendo del tipo de proceso. Para procesos que no son de tiempo real (los de usuario), se emplea `SCHED_NORMAL`, que es el esquema por defecto. Por el contrario, para procesos de tiempo real se dispone de los siguientes esquemas:

- `SCHED_FIFO`: es una planificación de tipo *FIFO* (*first-in-first-out*) en la que *no existen timeslices*, sino que cada proceso se ejecuta hasta que se suspende o explícitamente cede el procesador. Cuando existe más de un proceso `SCHED_FIFO`, se elige al de mayor prioridad (menor nivel).

- `SCHED_RR`: es como `SCHED_FIFO` salvo que sí existen *timeslices* y por tanto, cuando un proceso lo consume, este pasa al final de la cola de su nivel de prioridad y se elige al siguiente para su ejecución (planificación tipo *round-robin*). Procesos con este esquema de planificación tienen menos prioridad que los `SCHED_FIFO` y por tanto pueden ser expropiados por estos últimos.

El esquema de planificación de cada proceso se almacena en el campo `policy` de la estructura `task_t` (ver sección 2.1.1, página 12).

Cuando un proceso tiene un esquema de planificación de tiempo real, el planificador no emplea Prioridades Dinámicas(ver sección 2.4.1, página 38), sino que opera únicamente con el valor del campo `prio` (sin modificarlo) e ignora el de `static_prio` (de hecho, la llamada al sistema `nice()` no tiene ningún efecto sobre este tipo de procesos). Con esto se asegura que se respetará de forma estricta su nivel de prioridad.

### Cambiando las prioridades de los procesos de Tiempo Real

Linux proporciona una serie de Llamadas al Sistema para poder modificar y consultar la prioridad y el esquema de planificación de los procesos de tiempo real (`nice()` sólo afecta a los procesos de usuario). Estos métodos se resumen en la tabla 2.3.

Al igual que ocurre con `nice()`, utilizada para los procesos de usuario, en estas Llamadas al Sistema los valores de prioridad devueltos o introducidos como parámetros no son los internos del planificador (los Niveles de Prioridad), sino una conversión al rango [1..`MAX_USER_RT_PRIO-1`], donde 1 indica un proceso con la menor prioridad y `MAX_USER_RT_PRIO-1` el de mayor. El resultado de dicha conversión se almacena en el campo `rt_priority` de la estructura `task_t` de cada proceso y sólo vale 0 si ese proceso se ejecuta con el esquema de planificación `SCHED_NORMAL`.

`MAX_USER_RT_PRIO` es una constante definida en `include/linux/sched.h` y por defecto vale lo mismo que `MAX_RT_PRIO` (100). Sin embargo, si la primera tiene un valor menor, permite reservar los primeros niveles de prioridad a los *kernel threads* (procesos que sólo se ejecutan en modo *kernel*).

Para más detalles sobre estas llamadas al sistema, se puede consultar las páginas del Manual de Programación de Linux.

Cuadro 2.3: Llamadas al Sistemas para la planificación en tiempo Real

| *Llamadas al sistema* | *Descripción* |
|---|---|
| `sched_setscheduler()` | Establece el esquema de planificación y la prioridad de un proceso. |
| `sched_getscheduler()` | Devuelve el esquema de planificación de un proceso. |
| `sched_setparam()` | Establece la prioridad de un proceso de tiempo real. |
| `sched_getparam()` | Devuelve la prioridad de un proceso de tiempo real. |
| `sched_get_priority_max()` | Devuelve la mayor prioridad para un esquema de planificación. |
| `sched_get_priority_min()` | Devuelve la menor prioridad para un esquema de planificación. |

## 2.5. Otros aspectos

### 2.5.1. Cambio de Contexto

El cambio de contexto es la acción por la cual se cambia el proceso que está en ejecución por otro. Esto incluye guardar el estado de los registros de la CPU y cargar un nuevo estado, y cambiar el mapa de la memoria virtual. Esta operación es bastante costosa en tiempo y por tanto se intenta realizar lo menos posible.

En el planificador de Linux, esta labor se lleva a cabo en el método `schedule()` en tres etapas:

Primero se ejecuta la macro `prepare_arch_switch()` para que realice ciertas labores previas al cambio de contexto, tales como el adquirir un *lock* para proteger al proceso que va a entrar en ejecución. Estas labores sólo son necesarias para ciertas arquitecturas, como por ejemplo, ARM, MIPS, IA64 y SPARC. Para el resto, el comportamiento por defecto de esta macro es no hacer nada.

Posteriormente se invoca a la función `context_switch()` para que realice el cambio del mapa de memoria virtual y el estado de los registros a través

de las macros `switch_mm()` (definida en `include/asm-*/mmu_context.h`) y `switch_to()` (definida en `include/asm-*/system.h`) respectivamente. A partir de este punto, el resto del método `schedule()` es ejecutado por el nuevo proceso.

Finalmente se establece una barrera para sincronizar a los procesos y se llama a `finish_task_switch()`. Esta función invoca a `finish_arch_switch()` para que, en las arquitecturas que requerían labores previas al Cambio de Contexto, se realicen las correspondientes operaciones posteriores a dicho cambio (por ejemplo, liberar el *lock* del proceso).

### 2.5.2. Suspender y despertar a un proceso

Linux, como cualquier otro sistema operativo *multitarea*, suspende a un proceso cuando este no puede continuar ejecutándose, permitiendo así que otra tarea se ejecute en su lugar y no se desperdicie tiempo de proceso ni el resto de los recursos del sistema.

Un proceso puede suspenderse por muchas razones. La más común es mientras espera a que finalice una operación de E/S (lectura/escritura en disco o que el usuario pulse una tecla), aunque también es posible que lo haga porque no puede acceder a algún recurso o sección crítica del código ya que está protegida por un *lock*. En cualquier caso, siempre se suspende *a la espera de que se produzca alguna clase de evento*.

#### Estados de los procesos suspendidos

Al suspender a un proceso, este pasa a un estado especial en el que no puede ser ejecutado. Gracias a esto, se impide que el planificador seleccione a procesos que no pueden o no deben ejecutarse, o peor aún, que el proceso suspendido tenga que utilizar *espera activa* (un bucle que no hace nada, pero que mantiene al procesador "ocupado") hasta que se produzca el evento.

En realidad existen dos estados en los que un proceso puede estar suspendido, `TASK_UNINTERRUPTIBLE` y `TASK_INTERRUPTIBLE`. En el primero, el proceso se suspende ignorando cualquier señal hasta que se produzca el evento por el que espera. Por el contrario, en el segundo, el proceso puede ser despertado prematuramente para atender a alguna señal, por ejemplo, la señal `SIGTERM` enviada con el comando `kill`.

### Suspender a un proceso

Cuando se suspende a un proceso, este pasa a una *Cola de espera*. Se trata de una simple lista de tareas que esperan a que ocurra algún evento, de manera que cuando este sucede, el código que lo controla despierta a todos los procesos de esa lista. Estas colas de espera se representan en el kernel mediante el tipo `wait_queue_t` definido en `include/linux/wait.h`.

Generalmente, un proceso se suspende al ejecutar alguna llamada al sistema. En esa llamada suelen seguirse los siguientes pasos ([20] y [21]):

- Declara la cola de espera de forma estática mediante la macro `DECLARE_WAITQUEUE()`, o bien, la inicializa de forma dinámica a través de la función `init_waitqueue_head()`.
- Añade el proceso a la cola utilizando el método `add_wait_queue()` (definido en `kernel/fork.c`). Esta cola despertará a cualquier proceso que esté en ella cuando el código que produzca o controle el evento llame a la macro `wake_up()` (definida en `include/linux/wait.h`).
- Cambia el estado del proceso, según su naturaleza, a `TASK_UNINTERRUPTIBLE` o `TASK_INTERRUPTIBLE`.
- Entra en un bucle en el cual, si no se cumple la condición por la que se espera, invoca al método `schedule()` para que seleccione a otro proceso. Tal y como se explicó en la sección 2.2 (página 30), en `schedule()`, si el estado del proceso es `TASK_INTERRUPTIBLE` y tiene una señal pendiente, vuelve a entrar en ejecución para tratarla; mientras que si es `TASK_UNINTERRUPTIBLE`, se invoca a `deactivate_task()` para que lo saque de la *runqueue*.
- Cuando el proceso es despertado, vuelve a evaluar la condición y si no se cumple, repite el paso anterior.
- Finalmente, cuando el proceso es despertado y se cumple la condición, sale del bucle y se elimina de la cola de espera mediante la función `remove_wait_queue()` (definida en `kernel/fork.c`).

### Despertar a un proceso

Como se explicó anteriormente, cuando se produce el evento por el que espera un proceso, el código que controla dicho evento ejecuta la macro

`wake_up()` (definida en `include/linux/wait.h`) sobre la Cola de Espera asociada a ese evento. Esta macro es una simple llamada a la función `__wake_up()` (en `kernel/sched.c`), que a su vez llama a `__wake_up_common()` para que recorra la *waitqueue* y despierte a cada proceso. Esto se hace invocando para cada tarea a la función de despertamiento que tenga asociada la cola (campo `func` de la estructura `wait_queue_t`). Por defecto, dicha función es `default_wake_function()`, que es un simple envoltorio para `try_to_wake_up()` (que es la que realmente despierta al proceso).

La función `try_to_wake_up()`, después de hacer algunas comprobaciones iniciales, intenta despertar al proceso en la CPU que ejecuta esta función (que no tiene por qué ser la misma donde se ejecutó la tarea por última vez). Para ello se tienen que cumplir las siguientes condiciones:

- Esta CPU debe estar permitida para este proceso.
- No debe darse el caso de que esta CPU tenga ya mucha carga de trabajo y la del proceso, casi ociosa.
- Debe existir un Dominio de Planificación al que pertenezcan ambas CPUs que, o bien tenga activado el flag `SD_WAKE_AFFINE` (ver sección 2.1.4, página 24) y que el proceso ya no tenga datos válidos en la cache de su procesador (que sea *cache-cold*), o bien, que tenga activado el flag `SD_WAKE_BALANCE` y exista un desequilibrio.

Si no se cumple alguna de estas condiciones, se intenta despertar el proceso en su CPU (en la última en la que se ejecutó).

Una vez elegida la CPU (ya sea la del proceso o la actual), si está definida la variable de compilación `ARCH_HAS_SCHED_WAKE_IDLE`, se intenta buscar otro procesador del mismo dominio que esté vacío. Esto se realiza con la función `wake_idle()`, la cuál va invocando a `idle_cpu()` para averiguar si esa CPU no tiene procesos qué ejecutar.

Posteriormente, se activa el proceso mediante la función `activate_task()` la cual lo inserta en la nueva *runqueue* (actualizando su prioridad en función del tiempo que pasó suspendido). Además. se solicita una expropiación si tiene mayor prioridad que la tarea que se está ejecutando, y finalmente, se marca su estado como `TASK_RUNNING`.

### 2.5.3. Ceder el procesador

Linux proporciona la llamada al sistema `sched_yield()` como un mecanismo para que el proceso que está en ejecución, pueda ceder el procesador a otras tareas. El método `sys_sched_yield()` mueve al proceso del Array de Prioridad Activo al de Expirados. Gracias a esto, se consigue que el procesos no se ejecute "durante algún tiempo". Sin embargo, si el proceso es de Tiempo Real, entonces no se mueve de Array, sino que simplemente se pone al final de la lista de los procesos de su mismo nivel de prioridad. Al final del método, se invoca a `schedule()` para que seleccione a otro proceso y realice el Cambio de Contexto.

En algunas partes del código del planificador, se llama a la función `yield()`, pero lo único que hace es comprobar que la tarea está en estado `TASK_RUNNING` y después llama a `sched_yield()`.

### 2.5.4. Depurando el planificador

Programar en el kernel Linux, puede convertirse en una tarea bastante compleja, es especial porque cualquier fallo puede hacer que todo el sistema se bloquee. Por ello existen una serie de funciones que intentan ayudar en el proceso de depuración. Algunas de ellas se describen a continuación:

- `printk()`: Es equivalente a la función `printf()` de la biblioteca de C. Sin embargo, en esta función se puede especificar un *loglevel*, el cual permite decidir al kernel si debe imprimir el mensaje por consola o basta con escribirlo en el log del sistema. Los *loglevels*, ordenados del más importante al menos crítico, se pueden ver en la tabla 2.4 ([20]).

- `BUG_ON(condition)`: esta macro, permite hacer un volcado de la pila (`dump_stack()`) cuando se cumple `condition`. En realidad, es una simple llamada a `BUG()`, que es la que realmente hace el volcado. Ambas macros están definidas en `include/asm-*/bug.h`.

- `panic()`: parecida a las anteriores, pero se reserva para errores críticos, ya que, además del mensaje de error, detiene el sistema.

Además, existen otros métodos, específicos del planificador, que muestran información adicional para distintos aspectos del mismo:

**Cerrojos** :

Cuadro 2.4: *loglevels* del kernel

| *loglevel* | *Descripción* |
|---|---|
| KERN_EMERG | Condición de emergencia |
| KERN_ALERT | El problema requiere una acción inmediata |
| KERN_CRIT | Condición crítica |
| KERN_ERR | Condición de error |
| KERN_WARNING | Advertencia |
| KERN_NOTICE | Notificación de alguna condición |
| KERN_INFO | Información |
| KERN_DEBUG | Depurando |

Al definir `CONFIG_DEBUG_SPINLOCK_SLEEP`, se puede verificar a través de la macro `might_sleep()`, si es seguro obtener un *lock*, es decir, si la expropiación y las interrupciones están activadas. Es importante, ya que en caso contrario, se podría producir un interbloqueo.

**Expropiación** :

Es posible comprobar si se produce un desbordamiento al modificar el contador de cerrojos (`preempt_count()`) si se define la variable de compilación `CONFIG_DEBUG_PREEMPT`. En este caso, no se emplean las macros `add_preempt_count()` y `sub_preempt_count()`, definidas en `include/linux/preempt.h`, sino las funciones de `kernel/sched.c`, que son las que realizan esta comprobación.

**Procesos** :

Se puede ver el estado del sistema mediante el método `show_state()`, el cual recorre todos los procesos e invoca, para cada uno de ellos, a `show_task()`. Este método muestra información como el PID, el estado del proceso, el PPID, los PID de procesos hermanos (con un padre común), etc.

**Dominios de planificación** :

Definiendo la variable de compilación `SCHED_DOMAIN_DEBUG`, el método `sched_domain_debug()` muestra toda la jerarquía de los Dominios de Planificación en el sistema.

### 2.5.5. Estadísticas del planificador

Para facilitar la toma de medidas, por ejemplo en la ejecución de *benchmarks*, se han añadido al planificador una serie de contadores y estructuras que almacenan información acerca de la mayoría de los eventos y acciones que se producen. Estas estadísticas se registran por niveles siguiendo las principales estructuras de datos del planificador. Para activar este sistema de recopilación de datos, es necesario tener definida la variable de compilación `CONFIG_SCHEDSTATS`. A continuación se describe cada uno de los niveles de estadísticas:

#### Estadísticas a nivel de proceso

Para recopilar estas estadísticas existe un campo en la estructura `task_t` denominado `sched_info` el cual almacena toda la información. Este campo es de tipo `struct sched_info` y se detalla a continuación:

```
struct sched_info {
    unsigned long cpu_time;     // Tiempo que ha estado ejecutándose en esta CPU.
    unsigned long run_delay;    // Tiempo que ha estado esperando en una 'runqueue'.
    unsigned long pcnt;         // Número de 'timeslices' que se ha consumido en esta CPU
    unsigned long last_arrival; // Última vez que se ejecutó en una CPU
    unsigned long last_queued;  // Última vez que fue insertado en la 'runqueue'
};
```

Esta información es actualizada en el método `sched_info_switch()` cuando se produce un cambio de contexto.

#### Estadísticas a nivel de CPU

Para este nivel, se han incluido variables en la estructura `struct runqueue` que contabilizan los eventos que ocurren en todas las CPUs, tales como ceder el procesador, despertar un proceso y la planificación de tareas. A continuación se muestran estas variables:

```
struct runqueue {
    ...
    /* Estadísticas al ceder el procesador: */
    unsigned long yld_both_empty; // Número de veces que los arrays de prioridad quedaron
                                  // vacíos
    unsigned long yld_exp_empty;  // Número de veces que el array de expirados quedó vacío
    unsigned long yld_act_empty;  // Número de veces que el array de activos quedó vacío
    unsigned long yld_cnt;        // Número de veces que se ha llamado a sched_yield()

    /* Estadísticas de planificación: */
    unsigned long sched_switch;   // Número de veces que se han intercambiado los arrays
                                  // de prioridad
    unsigned long sched_cnt;      // Número de veces que se a llamado a schedule()
    unsigned long sched_goidle;   // Número de veces que fue seleccionado el proceso ocioso
```

```
    /* Estadísticas al despertar un proceso: */
    unsigned long ttwu_cnt;       // Número de veces que se ha llamado a try_to_wake_up()
    unsigned long ttwu_local;     // Número de veces que se ha despertado a un proceso en
                                  // esta CPU

    struct sched_info rq_sched_info; // Estadísticas de latencia

};
```

Como se puede observar, también se ha incluido el campo `rq_sched_info` de tipo `struct sched_info`. Este campo recopila las mismas estadísticas que en el caso de los procesos, salvo que aquí se contabilizan los valores totales (para todos las tareas que han estado en esa cola).

Para incrementar estos contadores, se emplean las macros `schedstat_inc()` y `schedstat_inc()` definidas en `kernel/sched.c`.

### Estadísticas a nivel de Dominio de Planificación

Al igual que con las Colas de Procesos, en la estructura `struct sched_domain`, se han incluido variables para contabilizar eventos propios de este nivel, como es el equilibrado de carga. Estos contadores son los siguientes:

```
struct sched_domain {
    ...
    /* Estadísticas en load_balance(): */
    unsigned long lb_cnt[MAX_IDLE_TYPES];        // Número de llamadas a load_balance()
    unsigned long lb_failed[MAX_IDLE_TYPES];     // Número de veces que no se consigue mover
                                                 // los procesos.
    unsigned long lb_balanced[MAX_IDLE_TYPES];   // Número de veces que se ha conseguido
                                                 // equilibrar la carga
    unsigned long lb_imbalance[MAX_IDLE_TYPES];  // Número de veces que se encuentra con un
                                                 // desequilibrio
    unsigned long lb_gained[MAX_IDLE_TYPES];     // Número de procesos que fueron migrados
    unsigned long lb_hot_gained[MAX_IDLE_TYPES]; // Número de procesos con datos en la cache
                                                 // que fueron migrados
    unsigned long lb_nobusyg[MAX_IDLE_TYPES];    // Número de veces que no se ha encontrado
                                                 // un grupo con mucha carga
    unsigned long lb_nobusyq[MAX_IDLE_TYPES];    // Número de veces que no se ha encontrado
                                                 // una 'runqueue' con mucha carga

    /* Estadísticas en active_load_balance() */
    unsigned long alb_cnt;                  // Número de Dominios que se intenta equilibrar
    unsigned long alb_pushed;               // Número de veces que se ha movido un proceso
                                            // a otra CPU del dominio
    unsigned long alb_failed;               // Número de veces que no se ha podido mover
                                            // un proceso a otra CPU del dominio

    /* Estadísticas en sched_exec(): */
    unsigned long sbe_attempts;             // Número de intentos de equilibrado en este dominio
    unsigned long sbe_pushed;               // Número de veces que se consigue un equilibrado en
                                            // este dominio

    /* Estadísticas en try_to_wake_up(): */
    unsigned long ttwu_wake_remote;         // Número de veces que se intenta despertar en otra CPU
```

```
    unsigned long ttwu_move_affine;         // Número de veces que se mueve un proceso a otra CPU
                                            // por afinidad
    unsigned long ttwu_move_balance;        // Número de veces que se mueve un proceso a otra CPU
                                            // por equilibrado
};
```

La constante `MAX_IDLE_TYPES` es una de las pertenecientes al tipo enumerado `idle_type` que se muestra a continuación:

```
enum idle_type
{
    SCHED_IDLE,     // La cola está sin procesos que ejecutar
    NOT_IDLE,       // La cola tiene procesos que ejecutar
    NEWLY_IDLE,     // La cola de procesos se está quedando sin procesos que ejecutar
    MAX_IDLE_TYPES  // Número de tipos IDLE
};
```

De esta manera, existen estadísticas del `load_balance()` para cada una de esas situaciones.

### Mostrando las estadísticas

Se pueden volcar las estadísticas a un fichero gracias a la entrada `schedstats` en el directorio `/proc`. Al leer dicha entrada, se ejecuta en `kernel/sched.c` el método `show_schedstat()`, el cual recorre los procesadores (sus *runqueues*) y los dominios de planificación, volcando los valores de todos los contadores.

Se puede consultar más información acerca de estas estadísticas en el fichero `Documentation/schedstats.txt`.

# Capítulo 3

# Planificación en SMT

## 3.1. Introducción

Con la tecnología *Hyper-Threading* tendríamos, en principio, una mejora del rendimiento debida a un mayor aprovechamiento de los recursos de ejecución del microprocesador. En realidad existen ciertas situaciones en las que el *Hyper-Threading* es especialmente efectivo a la hora de aprovechar recursos, pero también existen ciertas situaciones en las que el *Hyper-Threading* es perjudicial para el rendimiento. Todas estas situaciones se pueden favorecer o evitar, según convenga, realizando una correcta planificación de los procesos que se estén ejecutando en nuestro sistema. A lo largo de este capítulo se detallará cuáles son esas situaciones, lo que se ha intentado y lo que hay hecho ahora mismo para intentar aprovechar el Hyper-Threading o, al menos, que no nos suponga una pérdida de rendimiento.

## 3.2. Problemas del *Hyper-Threading*

Como ya se ha explicado, cuando el microprocesador está funcionando en modo *SMT* tenemos tres tipos de recursos:

- los duplicados
- los divididos
- los compartidos

Los recursos duplicados, que son los menos, no afectan al rendimiento, simplemente están en desuso cuando el microprocesador está en modo *ST*. Los recursos divididos suponen siempre una disminución del rendimiento,

pero se compensa si los dos hilos hacen trabajo. Con los recursos compartidos el problema es que, si se seleccionan para ejecución simultáneamente dos hilos que compiten por los mismos recursos, no tendremos la ventaja de la ejecución paralela, pero sí la desventaja de tener recursos divididos en ciertas partes del *pipeline*.

Como se puede ver, que el *Hyper-Threading* sea algo ventajoso o perjudicial para el rendimiento del sistema depende enteramente de la planificación que realicemos. Dos hilos bien planificados en paralelo podrían incrementar bastante el rendimiento de nuestra CPU con un incremento mínimo de precio que supone el *Hyper-Threading*, pero una mala planificación puede hacer que el rendimiento sea peor que teniendo una CPU sin *Hyper-Threading*.

## 3.3. Situaciones a evitar

Para sacarle el mayor partido a la tecnología *Hyper-Threading*, hay ciertas situaciones que el planificador debe evitar.

1. Espera activa: si uno de los hilos hace espera activa, está ocupando recursos que se dividen entre ambos procesadores lógicos y no hace trabajo útil. En esta situación tenemos un descenso grande del rendimiento.

2. Planificación de dos hilos simultáneos con las mismas necesidades funcionales: si dos hilos requieren utilizar las mismas unidades funcionales en ejecución, a pesar de estar ocupando recursos su trabajo se serializará y tenemos de nuevo una pérdida de rendimiento debido a los recursos divididos.

3. Planificación de dos hilos simultáneos que compitan por las mismas lineas de cache. Si los dos hilos están reemplazándose bloques el uno al otro, se pierde muchísimo rendimiento en dichos reemplazos.

## 3.4. Situaciones ventajosas

Existen otras situaciones en las que la tecnología *Hyper-Threading* da ventajas de rendimiento.

1. Planificación de hilos simultáneos cuyos recursos necesarios sean muy compatibles o totalmente compatibles, por ejemplo: un hilo intensivo en enteros y otro intensivo en coma flotante.

2. Planificación de dos hilos simultáneos que compartan memoria. Un hilo le haría *pre-fetching* de datos al otro, de forma que cuando el otro hilo fuese acceder al dato lo encontraría ya en cache.

3. Planificación de hilos simultáneos de los cuales uno de ellos o los dos, sufren muchas paradas por fallos de predicción o de cache. Mientras uno espera a que se resuelva su fallo el otro puede ir haciendo cosas, de forma que se ocultan las latencias del otro hilo. De esta forma se aumenta la productividad de la CPU, consigue un IPC más alto.

## 3.5. Primeros intentos

Los primeros intentos de hacer un planificador *Hyper-Threading aware* se realizaron en las versiones de desarrollo de Linux 2.6. Ingo Molnar[1] propuso unas características que debía cumplir un planificador para que se considerase *Hyper-Threading aware*:

- El equilibrado de carga pasivo (el que se dispara por tiempo) tiene que hacer el equilibrado entre CPUs físicas, no lógicas.
- El equilibrado de carga activo (cuando una CPU va a quedar ociosa), tiene que tener en cuenta que dos tareas que aparentemente están 2 CPUs distintas, podrían estar compartiendo una misma CPU física mientras que otra CPU física está ociosa.
- Cuando el planificador selecciona para ejecución una tarea, debería tratar de coger tareas de la misma CPU física (es decir, de una de sus CPUs hermanas) antes de sacar tareas de colas de otras CPUs físicas.
- La afinidad (*affinity*) en las tareas debería funcionar por procesadores físicos y no por procesadores lógicos.
- Las tareas que vayan a despertar en una CPU lógica que esté haciendo trabajo mientras alguna de sus hermanas está ociosa, deberían despertar en las CPUs hermanas que están ociosas.

Ingo Molnar propuso una solución que atajaba todos estos problemas de una sola vez: colas de ejecución compartidas por CPUs físicas. Hizo un parche contra la versión 2.5.31-BK-curr [22] con dicha solución, pero el parche se abandonó y nunca llegó a ser añadido a la rama principal del kernel.

[1] Ingo Molnar es el autor del planificador $O(1)$ de Linux 2.6, además de ser el actual encargado del planificador en dicha versión

En realidad, sus propuestas para un planificador *Hyper-Threading aware*, aunque efectivamente solucionaban algunos problemas (sobre todo en el equilibrado de carga para sistemas *SMP*), no eran del todo correctas. En realidad hay ocasiones en las que interesa que dos hilos estén en la misma CPU física aunque haya otra CPU física ociosa, como podría ser la situación en la que un hilo hace *pre-fetching* a otro hilo.

Ingo Molnar tampoco tuvo en cuenta el problema más serio de todos, la caída de rendimiento cuando hay conflicto de recursos. Esto se soluciona en parte en los sistemas *SMP* cuando se hace un equilibrado de carga entre CPUs físicas, pero en sistemas monoprocesador, o cuando en un sistema *SMP* hay muchas tareas listas para ejecución e importa cómo se emparejan en las CPUs físicas, su solución no es válida. Teniendo en cuenta que esta situación es la que más pérdida o mejora del rendimiento nos puede proporcionar, aunque su parche solucionaba algunos problemas menores de la planificación en *Hyper-Threading* no solucionaba los problemas más serios.

## 3.6. Situación actual

Actualmente existe un parche para el planificador que pretende mejorar el rendimiento en sistemas *SMT*. Lo que hace el planificador, a groso modo, es evitar que dos tareas con *nice* diferente concurran en una misma CPU física durante más de un cierto porcentaje de tiempo, enviando a dormir a la de mayor *nice* (menor prioridad). Utilizando esta técnica se han obtenido resultados experimentales de una mejora del 5 %. El problema de este parche es que deja en manos del programador y el administrador de sistemas asignar los valores de *nice* adecuados a cada proceso para optimizar el rendimiento, en lugar de hacer una planificación *Hyper-Threading aware* transparente a los administradores y programadores del sistema.

### 3.6.1. El planificador SMT de Linux

Para la planificación descrita en 3.6, se realizan las siguientes modificaciones:

- Define la función `int cpu_and_siblings_are_idle(int cpu)`, que recorre todo el mapa de CPUs hermanas comprobando que se encuentran todas ociosas. La utiliza desde:
  - `int can_migrate_task(task_t *p, runqueue_t *rq, int this_cpu, struct sched_domain *sd, enum idle_type idle)`

para saber si puede migrar una tarea a una determinada CPU física por estar ociosa. En el caso particular de *SMT* no vale con que la CPU hermana esté ociosa, tienen que estarlo todas las hermanas.

- `void active_load_balance(runqueue_t *busiest_rq, int busiest_cpu)` que su vez es ejecutado por los hilos de migración (2.3), lo utiliza para saber si una CPU física está ociosa, evitando el desequilibrio entre CPUs físicas. Procura tener ocupadas primero todas las CPUs físicas y luego las lógicas.

- Define la función `int wake_priority_sleeper(runqueue_t *rq)` que replanifica el proceso *idle*, de forma que si hay tareas que se enviaron a dormir por razones de prioridad se despertarán. Se utiliza en `void scheduler_tick(void)`, que es llamada desde el *timer* y desde *fork* cuando se cambia el *timeslice* del padre, y actualiza los *timeslices* de las tareas.

- Define la función `void wake_sleeping_dependent(int this_cpu, runqueue_t *this_rq)`, encargada de despertar todas las tareas dormidas por razones de prioridad en las CPUs lógicas que componen una CPU física. Se llama desde la la función principal del planificador (`void __sched schedule(void)`, ver capítulo 2) cuando una cola de ejecución se queda sin tareas para correr.

- Define la función `int dependent_sleeper(int this_cpu, runqueue_t *this_rq)`. Se llama desde la función principal de planificación para preguntar si puede poner a ejecutar alguna tarea de *rq* en la CPU dada, si no puede se selecciona la tarea *idle* para dejar la CPU libre a alguna de las tareas más prioritarias de las CPUs hermanas.

La función selecciona la tarea más prioritaria de la cola de ejecución y compara su *timeslice* con el de las tareas que estén corriendo en las CPUs hermanas, si le queda un *timeslice* de menos de un 75 % del que le queda a la tarea hermana, ni esa ni ninguna otra tarea de dicha cola de ejecución podrá entrar a la CPU. Además, esta función se encarga de replanificar aquellas tareas hermanas en ejecución cuya prioridad sea menor (es decir, que su *timeslice* sea menor del 75 %) que la de la tarea seleccionable. De ese modo se evita que una tarea de baja prioridad moleste a las tareas más prioritarias.

- Define el vector `sched_group sched_group_cpus[NR_CPUS]` con los grupos de planificación a los que pertenece cada CPU. Inicializa este vector asignando a cada CPU el grupo que devuelve la función `int __devinit cpu_to_cpu_group(int cpu)` (que es la propia CPU así que `sched_group_cpus[i] = i`). Este vector es el que utiliza para inicializar los grupos de planificación de cada dominio de planificación (ver 2.1.4).
- Define la función `int __devinit cpu_to_phys_group(int cpu)` que devuelve una CPU representante de la CPU física a la que está asociada *cpu*. La CPU que devuelve como CPU física es la primera de todas las hermanas, en el caso de un solo procesador con *Hyper-Threading*, tanto a la CPU 0 como a la CPU 1 les corresponderían la CPU física 0.

Además, para las arquitecturas *NUMA+SMT*, al inicializar los grupos de planificación comprueba que no hay CPUs lógicas de una misma CPU física asignadas a distintos nodos *NUMA* con la función
`void __devinit arch_init_sched_domains(void)`

El parche funciona de forma que cada vez que se va a seleccionar una nueva tarea para ejecución, si hay alguna tarea más prioritaria tal que el 75 % de su *timeslice* (valor que se ha obtenido experimentalmente) sea mayor que el suyo, esa tarea no se manda a ejecutar. En su lugar se envía el proceso *idle*, que pasa de modo *SMT* a modo *ST* cediendo el control de la CPU física en exclusiva al hilo más prioritario. Si por el contrario, la tarea seleccionada es más prioritaria que las que estén corriendo en las CPUs hermanas, las tareas de dichas CPUs cuyo *timeslice* sea menor de un 75 % del *timeslice* de la tarea seleccionable se replanifican.

# Capítulo 4

# Contadores hardware

Los contadores hardware están presentes en los procesadores de Intel desde la aparición de los Pentium. Este mecanismo de monitorización consiste en la incorporación de un conjunto de contadores específicos denominados *MSR* en la arquitectura IA-32, los cuales ocupan un área despreciable dentro del chip. Estos contadores permiten seleccionar el parámetro que va a ser monitorizado. La información obtenida de estos contadores puede ser gran ayuda para optimizar el sistema.

El uso de estos contadores en teoría es algo muy práctico y útil, pero debido a numerosos factores, estos simplemente se utilizan para tomar medidas y realizar *benchmark*. Con estas medidas se pueden ver los cuellos de botella que genera el código e intentar modificar este. Con este proyecto se intenta dar un paso más, integrando estas medidas dentro del propio planificador.

Por desgracia, los contadores que incorporan los procesadores de Intel es "hardware de segunda" como lo denomina Brinkley Sprunt, el jefe de desarrollo de los contadores en los Pentium 4. En su conferencia del 11º HPCA (*International Symposium on High Performance Computer Architecture*) celebrado del 12 al 16 de Febrero de 2005 en San Francisco, EEUU, expone todo lo que pueden llegar a proporcionar, quejándose a su vez de como Intel no llega incluso a validar ese hardware y de la poca documentación proporcionada.

Gracias a los contadores hardware, el planificador se ha modificado para adecuarlo a las necesidades de los procesadores con *Hyper-Threading*, posibilitando que la planificación realizada sea "inteligente" y se adapte a lo que realmente está pasando en ese instante en el procesador. En este caso y al ser

un planificador específico para procesadores con tecnología *Hyper-Threading*, se tendrá en cuenta qué está pasando con los recursos considerados críticos (aquellos que se comparten entre los dos procesadores lógicos). La utilización de los contadores hardware también sería aplicable a planificadores menos genéricos y a otros procesadores no fabricados por Intel, ya que tanto los procesadores fabricados por IBM y AMD también cuentan con su sistema de contadores.

El capítulo queda dividido en dos bloques, el primero abarca los primeros puntos, hasta 4.2 incluido, en el que se describe como son y funcionan los contadores hardware para la familia de procesadores Pentium 4 y Xeon. Solamente se ha desarrollado para estos procesadores porque son los que pueden soportar *Hyper-Threading*, quedando el resto de familias de procesadores fuera del alcance de este proyecto. Se puede consultar más información sobre el sistema de contadores hardware del resto de procesadores de Intel en [26]. Para los microprocesadores de otros fabricantes, habrá que remitirse a los manuales de desarrollador para el procesador en concreto proporcionado por el fabricante. En el punto 5.3 se explican las funciones añadidas al fichero `htaboot.c` que permiten la configuración y lectura de los contadores de rendimiento.

## 4.1. Contadores de rendimiento (*Performance Counters*)

Los mecanismos para monitorizar el rendimiento de los procesadores de las familias Pentium 4 y Xeon es diferente a los utilizados en los la familia P6 y los tradicionales Pentium, siendo incompatibles unos con otros. Los Pentium 4 y Xeon disponen de contadores de rápida lectura. Estos procesadores pueden contar numerosos eventos que suceden en el procesador, aunque no todos son compatibles y no pueden ser medidos a la vez, debido a la limitación que impone la lógica. El conjunto de contadores para la monitorización del rendimiento está formado por:

- El IA32_MISC_ENABLE MSR, que indica la capacidad de un procesador IA-32 para monitorizar el rendimiento y la presencia de mecanismos PEBS (es un modo de monitorización "preciso" que se detallará en el apartado 4.1.4).
- El control de selección de eventos se lleva a cabo con el registro ESCR. El valor que se pasa a dicho registro varía en función de la familia de procesadores.
- La lectura del número de eventos producidos se realiza de los registros MSR, de los que cada microprocesador dispone de 18.
- A su vez, cada uno de estos MSRs tiene asociado un registro CCCR, denominado Registro de Control de Configuración y establece el método específico o el estilo de conteo. Hay por tanto 18 de estos registros.
- Un área de depuración denominada DS (*debug store*) donde se salvan los registros PEBS.
- El IA32_DS_AREA MSR que establece la dirección del área del punto anterior.
- La DS lleva asociados unos *flags* para indicar que procesadores tienen habilitado los mecanismos de la DS. Es el bit 21 y es devuelto por la instrucción "CPUID".
- El IA32_PEBS_ENABLE MSR que activa la lógica PEBS y permite el uso de etiquetado y conteo de eventos *at-retirement*.
- Un conjunto predefinido de eventos y métricas para simplificar la monitorización de ciertos eventos.

El etiquetado mencionado anteriormente es un sistema que permite marcar o etiquetar ciertas instrucciones para contabilizar eventos que se dan solamente con estas instrucciones marcadas, por ejemplo las instrucciones en punto flotante. Este etiquetado se lleva a cabo mediante la configuración del registro CCCR correspondiente. Los eventos *at-retirement* son los eventos que ocurren cuando se está retirando instrucciones del ROB. La relación que hay entre este tipo de eventos y el etiquetado de instrucciones permite monitorizar, por ejemplo el número de instrucciones de un cierto tipo retiradas (o ejecutadas) en un programa.

Los tipos de eventos que pueden ser monitorizados se dividen en dos clases:

- Eventos *non-retirement*, son los eventos que suceden durante la ejecución de la instrucción, como transacciones en el bus o con la cache.

- Eventos *at-retirement* son los eventos que se cuentan cuando se retira la instrucción, lo que permite capturar el estado de la máquina, proporcionando gran versatilidad en el conteo de eventos. El mecanismo de conteo de estos eventos incluye lógica para etiquetar $\mu$ops y poder contar un determinado evento que se da en un cierto tipo de instrucciones. El etiquetado permite distinguir entre los eventos que se producen en la fase *commit* y los que ocurren durante la ejecución. Ciertos eventos que se dan en la fase de ejecución implican la cancelación de la ejecución de la instrucción (debido a la ejecución especulativa) como pueden ser fallo en la predicción de saltos.

Los procesadores Pentium 4 y Xeon, soporta tres modelos de monitorización que se pueden combinar con las dos clases de eventos descritas antes. El primero de los modelos se puede usar con ambas clases, y el último solamente con los eventos *at-retirement*:

- *Event counting*. Se configura un contador de rendimiento para contar uno o mas tipos de eventos. Mientras el contador está contando, el software lee el contador seleccionado en intervalos determinados por el número de eventos que se han producido entre intervalos.

- *Non-precise event-based sampling*. Un contador de rendimiento se configura para contar uno o varios eventos y para generar una interrupción cuando el contador se desborda. Cuando el contador desborda, el procesador genera una interrupción de monitorización de rendimiento o PMI. Una vez ejecutada la rutina de tratamiento de dicha interrupción, se

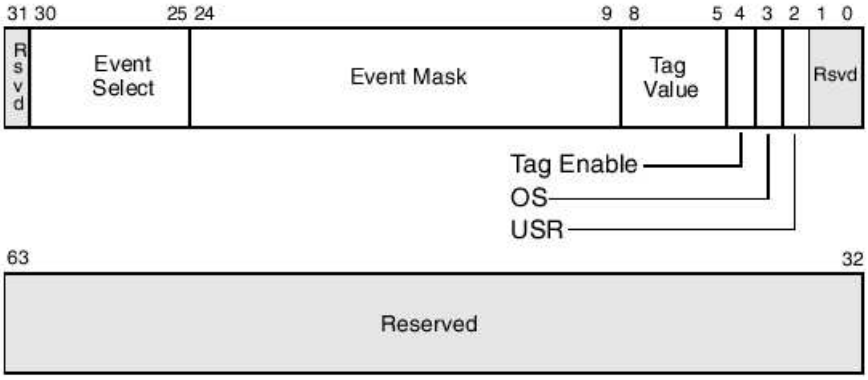


Figura 4.1: Registro de selección de evento y control (ESCR) para Pentium 4 y Xeon

devuelve el puntero de instrucción o RIP, se resetean los módulos y el contador.

- *Precise event-base sampling* o PEBS. Este tipo de monitorización es similar al anterior, excepto que se usa un *buffer* en memoria para salvar el estado de la arquitectura del procesador cuando el contador se desborda. Este registro del estado proporciona información adicional. Como ya se ha comentado antes, este modelo solamente se puede usar con los eventos de la clase *at-retirement*.

### 4.1.1. Estructuras de los contadores

A lo largo de esta sección se detallarán las estructuras de las que está compuesto el sistema de contadores de rendimiento de los Pentium 4 y Xeon.

#### ESCR MSR's

Los 45 registros ESCR MSR's (ver tabla 15-2 de [26]) permiten seleccionar vía software eventos específicos para ser contados. Cada ESCR normalmente va asociado con un par de contadores, y cada contador tiene varios ESCR's asociados con él, permitiendo así que los eventos sean contados y seleccionados. En la figura 4.1 se puede ver un esquema de este tipo de registro. Los diferentes campos representan:

- *Flag USR*, bit 2. Cuando está activado, los eventos se cuentan cuando el procesador está en el nivel de privilegios (CPL) 1, 2 ó 3. Estos niveles son utilizados en las aplicaciones de usuario.

- *Flag OS*, bit 3. Cuando está activado se cuentan los eventos que se producen cuando el nivel CPL es 0. Este nivel de privilegios, normalmente está reservado al sistema operativo. Cuando este *flag* y el anterior están activados, se contarán los eventos que se producen en todos los niveles de privilegios.

- *Flag Enable*, bit 4. Activado cuando se permite el etiquetado de $\mu$ops para contar un eventos *at-retirement*. Cuando está a 0, está desactivado el etiquetado.

- *Tag Value field* o Valor de la Etiqueta, del bit 5 al 8. Como su propio nombre indica, selecciona un valor para la etiqueta utilizada.

- *Event Mask field* o Máscara de Evento, del bit 9 al 24. Esta máscara es la que permite seleccionar que evento contar de la clase de eventos seleccionada en el campo de selección de evento.

- *Event Select field* o Selección de Evento, del bit 25 al 30. Selecciona la clase de evento. Para seleccionar un evento de esta clase se pondrá el valor correcto en el *flag* de Máscara de Evento.

Cuando se configura un ESCR, en el campo de selección del evento se especifica la clase de evento que se quiere contar, como por ejemplo, los saltos retirados. Con la máscara de evento se selecciona el evento o los eventos de la clase seleccionada que se quieren contar. Por ejemplo, cuando se cuentan los saltos retirados, se pueden contar hasta cuatro eventos diferentes: saltos bien predichos y no tomados, saltos no tomados y mal predichos, saltos tomados y bien predichos y saltos tomados mal predichos. Por último, en función de los *flags* USR y OS se seleccionará si se cuentan los que se producen en las aplicaciones de usuario y/o en la ejecución del sistema operativo.

Los ESCRs se inicializan con todos sus bits a 0. Los *flags* y campos de estos registros se configuran escribiendo sobre ellos con la instrucción "WRMSR" (ver tabla 15-2 de [26] para ver las direcciones y [27] para ver la sintaxis de la instrucción).

Escribir en un registro ESCR MSR solamente configura el contador para que cuente un determinado evento, pero no inicia la cuenta. Faltaría configurar el CCCR, que junto a la configuración del ESCR activaría la cuenta del evento en el contador seleccionado.

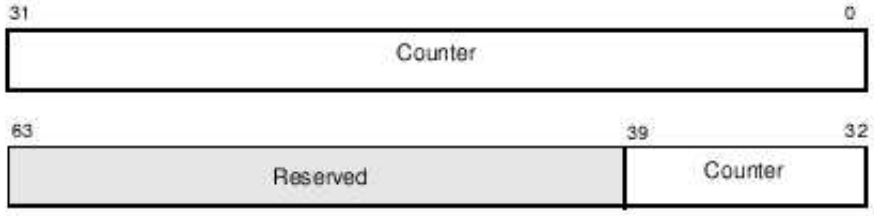


Figura 4.2: Contador de rendimiento en Pentium 4 y Xeon

### Contadores de rendimiento

Los contadores de rendimiento junto a los registros de configuración y control (CCCRs) son los que proporcionan las medidas de los eventos que se seleccionan en los ESCRs. Los Pentium 4 y Xeon tienen 18 contadores organizados en 9 pares. Cada pareja de contadores va asociada a un subconjunto particular de eventos y ESCRs. Para saber qué eventos pueden contarse con cada pareja hay que remitirse a la Tabla 15-6 de [26], al igual que para ver la clasificación de las parejas de contadores. Cada contador tiene 40 bits, como se puede ver en la figura 4.3. La instrucción "RDPMC" permite leer estos registros. Una de las peculiaridades de los Pentium 4 y Xeon, como ya se ha dicho antes es que permiten la lectura rápida de contadores. Esta lectura rápida consiste en leer los 32 bits menos significativos de estos registros. Esto es útil cuando se sabe de antemano que el evento que se va a contar no va a exceder el valor máximo y por tanto no va a producir un desbordamiento del contador. La instrucción "RDPMC" se puede usar en programas con cualquier privilegio y en el modo 8086-virtual, aunque se puede restringir su uso a nivel de privilegio 0 configurando el *flag* PCE del registro de control CR4.

La manipulación de los contadores sólo la puede llevar a cabo el sistema operativo ejecutándose en nivel de privilegios 0, mediante las instrucciones "RDMSR" y "WRMSR". Un sistema operativo seguro debe limpiar el *flag* PCE durante la inicialización del sistema y desactivar el acceso a los usuarios a los contadores hardware, y a la vez proveer a los usuarios de una interfaz que permita emular la instrucción "RDPMC".

En el caso de este proyecto, se pretende leer los contadores cada cierto intervalo de tiempo para modificar el comportamiento del procesador en función a lo leído. Por tanto será necesario también el poder escribir en los contadores. Para eso se utiliza la instrucción "WRMSR" con la que se configuran los contadores, como ya se verá cuando se comente el código.

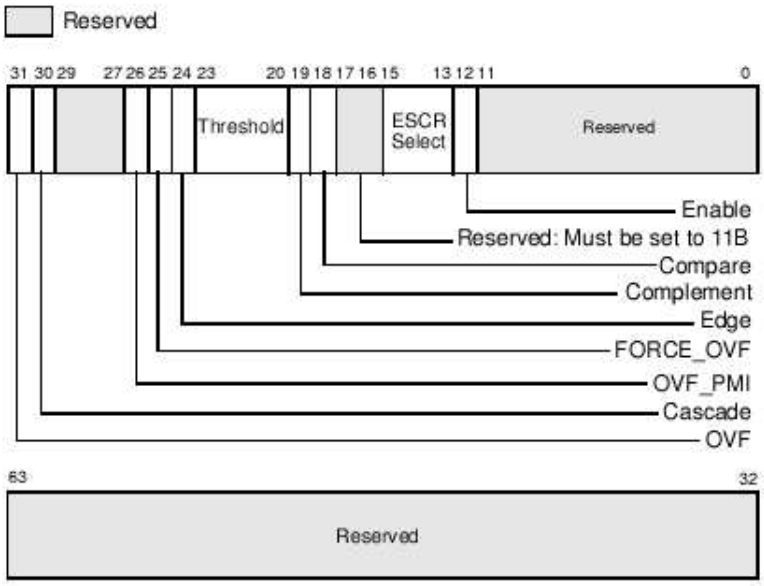


Figura 4.3: Registro de configuración (CCCR)

### CCCR MSRs

Cada uno de los 18 contadores de los que disponen los Pentium 4 y Xeon tiene un CCCR MSR asociado, que como ya se ha dicho lleva el control sobre qué evento se va a contar. En la figura 4.3 se pueden ver los campos de estos registros, que son puestos todos a 0 cuando el CCCR es reseteado:

- *Flag* de activación (*Enable flag*), bit 12. Cuando está seleccionado, está activa la cuenta. Cuando se borra o se hace un *reset* se pone a 0 y deja de contar.

- Selección de ESCR (*ESCR select field*), bits 13 al 15. Identifica el ESCR que se usará para seleccionar los eventos que se contarán en el contador asociado al CCCR.

- *Flag* de comparación (*Compare flag*), bit 18. Cuando está seleccionado se activa el filtro para el contador. Se puede filtrar el valor del contador mediante un umbral, complemento y *edge flags*.

- *Flag* de complemento (*Complement flag*), bit 19. Selecciona como comparar el valor del contador con el valor umbral. Cuando está seleccionado, los valores menores o iguales al umbral se entregan. En caso de

que no esté activado, son los valores mayores. Para más información sobre el filtrado de eventos se recomienda consultar [26].

- Umbral (*Threshold field*), bits 20 al 23. Campo para establecer un valor umbral con el qué comparar en los filtrados de eventos.

- *Edge flag*, bit 24. Cuando esta seleccionado activa la salida de la comparación. Sólo funciona cuando está activado el *flag* de comparación.

- *FORCE_OVF flag*, bit 25. Fuerza a que la señal de desbordamiento se active en cada incremento del contador.

- *OVF_PMI flag*, bit 26. Si está activado entonces se lanzará una interrupción de contador de rendimiento o PMI (*performance monitor interrupt*) cada vez que suceda un desbordamiento en el contador. Si está desactivado no se lanza PMIs.

- *Cascade flag*, bit 30. Con que este *flag* esté activado en uno de los contadores que forma la pareja, hace que los contadores funcionen en cascada, es decir que se alternen con otros cuando se produce desbordamiento.

- *OVF flag*, bit 31. Estará activado cuando el contador haya sufrido un desbordamiento.

Por tanto, el evento seleccionado y la máscara de evento del ESCR establecen la clase de evento que se va a contar y el tipo o tipos dentro de esta (se pueden contar varios eventos de la misma clase a la vez). Los *flags* OS y USR en el ESCR establecen el nivel de privilegios desde donde van a ser medidos. En el campo de selección de ESCR, que se encuentra en el CCCR se selecciona el ESCR, debido a que cada contador lleva asociado varios ESCRs. Después se puede configurar los filtros, con los *flags* de comparación, complemento y el *edge flag*.

### DS save area

La *DS save area* se utiliza para guardar la siguiente información:

- Registro de saltos. Cuando el *flag* BTS del MSR_DEBUGCTLA MSR está activado, se guarda un registro con el salto en el buffer de BTS situado en la DS, siempre que el salto sea tomado, interrumpido o sea detectada una excepción.

- Registro PEBS. Cuando un contador de rendimiento es configurado para PEBS (medida precisa), se almacena un registro PEBS cuando se produce un desbordamiento del contador. Este registro contiene el estado de la arquitectura del procesador (esto incluye el estado de los 8 registros, el registro EIP y el registro con los *flags* EFLAGS) en el momento del desbordamiento. Cuando la información ha quedado registrada, el contador automáticamente se resetea al valor establecido y comienza la cuenta de nuevo. Esta característica esta solamente disponible en los procesadores Pentium 4 y Xeon.

### 4.1.2. Programación para eventos *Non-Retirement*

Para poder medir los eventos hay que seguir una serie de pasos:

1. Seleccionar el evento o eventos que se quieren medir.
2. Para cada evento, seleccionar un ESCR compatible con esos eventos comprobando las restricciones. Esto se puede ver en la Tabla A-1 de [26].
3. Seleccionar el contador CCCR donde se va a contar a partir de los valores de ESCR.
4. Establecer los niveles de privilegios que se van a monitorizar en el ESCR.
5. Activar el conteo en cascada si se desea.
6. Opcionalmente configurar el CCCR para generar interrupciones PMI cuando el contador desborda. El APIC local debe estar preparado.
7. Comenzar el conteo del evento.

Para todos estos pasos, en [26], se detalla como conseguir los valores adecuados. Conseguir los valores con este método es algo tedioso, con lo que se buscó otra forma para conseguir estos valores. Para ello se utilizaron los programas "Perfctr" y "Brink & Abyss", como se detalla en la sección 5.3.

### 4.1.3. Medir eventos *at-retirement*

Medir eventos del tipo *at-retirement* significa medir solamente eventos que pueden darse en la fase de *commit*, ignorando por tanto todo el trabajo de especulación que lleva a cabo el procesador.

La microarquitectura *NetBurst* usada en los Pentium 4 y Xeon permite que se realicen muchas actividades especulativas para incrementar la efectividad y elevar el rendimiento, como puede ser la predicción de saltos. Los Pentium 4 y Xeon llevan el típico predictor de saltos con la dirección de estos y después decodifican y ejecutan la instrucción predicha, anticipándose al cálculo de la dirección real. Cuando se produce un fallo en la predicción, los resultados de las instrucciones que han sido decodificadas y ejecutadas por la rama del fallo de predicción, se cancelan.

Mediante *at-retirement* se pueden monitorizar eventos que previamente han sido etiquetados (*tagging*) y que son seleccionados por dicha etiqueta en la fase *commit*. Intel proporciona las siguientes definiciones para clasificar eventos e instrucciones:

- Instrucciones *Bogus*, *Non-Bogus* y *retired*. Con *Bogus*, Intel se refiere a las instrucciones o $\mu$ops que deben ser canceladas porque están en una rama de ejecución en la que se ha producido un fallo de predicción de salto. Con *Non-Bogus* y *retired* se refiere a las instrucciones o $\mu$ops que producen cambios en el estado de la arquitectura. Las instrucciones pueden ser *retired* o *Non-Bogus* pero nunca las dos a la vez.

- Etiquetado *Tagging*. Consiste en marcar $\mu$ops relacionadas con un evento particular que se quiere monitorizar. Se contará el número de instrucciones cuando sean retiradas. Durante la ejecución, se puede producir el evento mas de una vez por $\mu$ops, con lo que no es del todo exacto, ya que al final se miden las instrucciones y no el evento directamente. El etiquetado permite que una $\mu$op sea etiquetada una vez y esta mantendrá la etiqueta el tiempo en el que esté en el *pipe*, para ser contada al ser retirada. Se utiliza para métricas de rendimiento que se incrementan en uno por cada $\mu$op retirada.

- *Replay*. Para maximizar el rendimiento en las actividades más comunes, la microarquitectura *NetBurst* hace una planificación "agresiva" para poder ejecutar $\mu$ops antes de que se tengan garantías de la correcta ejecución. Cuando no se garantiza alguna de las condiciones necesarias para la ejecución de la $\mu$op, se produce un evento denominado "relanzamiento" (*reissued*). Para poder relanzar la $\mu$op los procesadores Pentium 4 y Xeon utilizan el mecanismo *replay*. Algunos ejemplos de *replay* son los fallos de predicción y violaciones de dependencias. En operaciones normales, algunos de los *replay*s producidos son comunes e inevitables. Un excesivo número de *replay*s indica que se está produciendo un problema de rendimiento.

- *Assist*. Cuando el hardware necesita ayuda con el microcódigo, la máquina le proporciona lo que se denomina "asistencia". Por ejemplo, cuando no han llegado los operandos necesarios para una operación de punto flotante. El hardware debe internamente modificar el formato de los operandos para que no se degrade el rendimiento.

### 4.1.4. *Precise Event-Based Sampling* (PEBS)

El DS de los Pentium 4 y Xeon permite recoger dos tipos de información para usarlos en depuración y optimización de programas. Estos son los registros PEBS y los BTS. Los BTS o *Branch Trace Store* son las trazas de saltos tomados, interrupciones y excepciones.

PEBS permite salvar el estado preciso de la arquitectura asociada con uno o más eventos. Esta información se guarda en un buffer que es una parte asignada del DS. Para usar este mecanismo, un contador tiene que ser configurado para permitir lanzar el desbordamiento. Cuando un contador desborda, el procesador copia el estado actual de los registros de propósito general, los registros EFLAGS y el contador de programa en un registro en el buffer situado en el DS. Después el procesador borra el contador y comienza a contar de nuevo. Cuando el buffer está casi lleno, se genera una interrupción, permitiendo que los registros sean salvados. No se trata de un buffer circular, para evitar que se sobreescriban datos.

Solamente soportan PEBS un determinado subconjunto de eventos *at-retirement* como son: *Execution_event*, *Front_end_event*, y *Replay_event*.

Para activar PEBS, el procesador debe soportarlo. Esto se puede observar viendo el resultado de ejecutar la instrucción "CPUID". Si está permitido, entonces son accesibles los siguientes campos:

- El flag PEBS_UNAVAILABLE en el registro IA32_MISC_ENABLE MSR, que indica si está habilitado PEBS (cuando está a 0).
- El flag de activación de PEBS (bit 24) en el IA32_PEBS_ENABLE MSR permite activar o desactivar PEBS.
- El IA32_DS_AREA MSR, que puede ser programado para que apunte al DS.


Figura 4.4: Registro de selección de evento y control (ESCR) para procesadores Pentium 4 y Xeon con *Hyper-Threading*

## 4.2. Contadores de rendimiento en *Hyper-Threading*

Para procesadores con tecnología *Hyper-Threading* la estructura de los registros utilizados para la monitorización de eventos incluye unas ligeras diferencias. Estas se deben a la presencia de dos procesadores lógicos dentro del mismo procesador físico. Así, Intel ha modificado estas estructuras para que el programador pueda decidir si medir los eventos de un procesador lógico, del otro, o incluso los de los dos a la vez. A continuación se detallan las estructuras que cambian, y qué es lo que cambia respecto a las vistas en el punto 4.1.1.

### 4.2.1. ESCR MSRs

El cambio respecto al ESCR visto en la figura 4.1 es que los bit 0 y 1 que estaban reservados se utilizan para establecer si se quiere medir los eventos producidos en el procesador lógico 1. Con el bit 0 (T1_USR) activado se contarán los eventos producidos en las aplicaciones del usuario, y con el bit 1 (T1_OS) activado se contarán los que se producen en el código del sistema operativo. El uso de estos dos bits, permite una gran variedad a la hora de medir eventos, ya que se puede elegir qué procesador se quiere medir o si son los dos (activando los dos a la vez) y a su vez ver en que parte de código se generan los eventos de cada procesador. El resto de los campos permanecen igual.

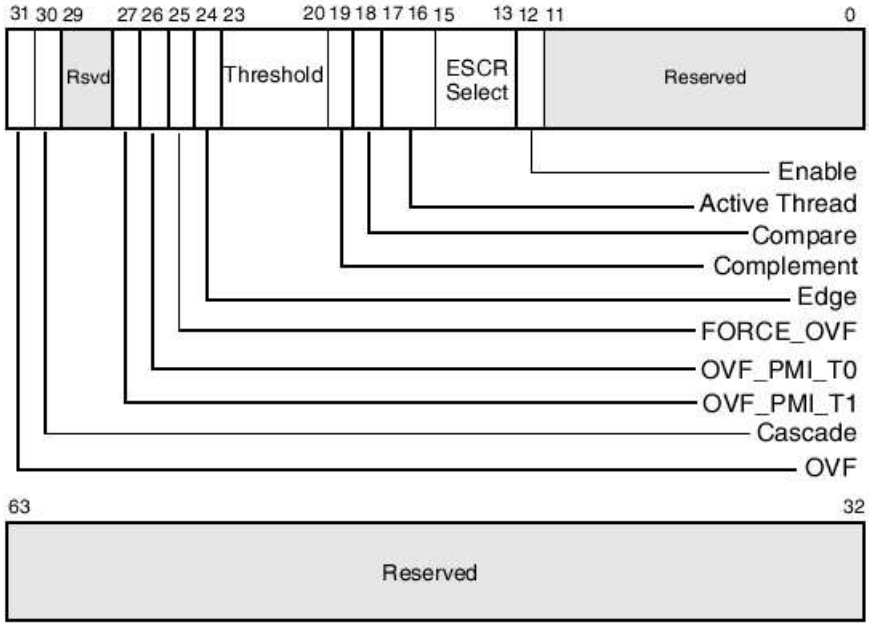


Figura 4.5: Registro de configuración (CCCR) para procesadores con HT

### 4.2.2. CCCR MSRs

Al igual que en el caso anterior, los campos añadidos provienen de los bits reservados que se podían en la figura 4.3:

- *Active Thread*, bits 16-17. Activa la cuenta dependiendo de que procesador lógico esté activo, es decir ejecutando un hilo. Este campo activa un filtro basado en el estado de los eventos (activo o inactivo) del procesador lógico. Los valores que puede tomar son los siguientes:

  **00** - Ninguno. Cuenta sólo cuando ninguno de los procesadores está activo.

  **01** - Sólo uno. Cuenta cuando solamente uno de los procesadores lógicos esta activo. Da lo mismo cuál sea.

  **10** - Ambos. Cuenta sólo cuando los dos procesadores lógicos están activos.

  **11** - Alguno. Cuenta cuando alguno de los lógicos se encuentre activo, incluyendo cuando los dos están activos.

- *OVF_PMI_T1*, bit 27. Si está activado entonces se lanzará desde el procesador lógico 1, una interrupción de contador de rendimiento o PMI

(*performance monitor interrupt*) cada vez que suceda un desbordamiento en el contador. Si está desactivado no se lanza PMIs.

- *OVF_PMI_T0*, bit 26. Este campo no se ha añadido, simplemente se ha renombrado. Su nombre anterior era OVF_PMI, y se refería a la única CPU disponible. Su función es la misma que la vista en 4.1.1 pero para el procesador lógico 1.

El resto de los campos mantienen su nombre y funcionalidad.

### IA32_PEBS_ENABLE MSR

En los procesadores con *Hyper-Threading*, PEBS se activa y gestiona mediante dos bits en el IA32_PEBS_ENABLE MSR, en concreto el bit 25 (ENABLE_PEBS_MY_THR) y el bit 26 (ENABLE_PEBS_OTH_THR). Estos bits no especifican el procesador lógico, sino que permiten activar PEBS para una secuencia de hilos de ejecución del mismo procesador lógico en el que se está ejecutando (de ahí el nombre del campo, mi hilo) o para el otro procesador lógico donde se estará ejecutando otro hilo.

Las restricciones sobre los eventos que soportan PEBS se siguen manteniendo, solo se puede usar con un subconjunto de eventos de la clase *at-retirement*: *Execution_event*, *Front_end_event*, y *Replay_event*.

# Capítulo 5

# Implementación

A lo largo de este Capítulo, se detalla como se ha implementado el planificador simbiótico HTAS (ver el prefacio). La planificación desarrollada a partir de las lecturas de los contadores, ha consistido en conseguir que el *Hyper-Threading* se "active" y se "desactive". Hasta encontrar una solución adecuada a la desactivación/activación del *Hyper-Threading* se ha pasado por cuatro "ideas" generales, de ahí que se hayan separado en varias secciones. Esto ha sido lo más problemático del proyecto, debido a la complejidad de Linux.

El entorno de desarrollo, depuración y pruebas ha sido un PC, equipado con un procesador Intel Pentium 4 a 3 GHz con *core Northwood*. No se ha utilizado ningún tipo de simulador, siendo obtenidos todos los resultados sobre máquinas reales, hasta ahora las pocas pruebas que se han hecho con planificadores simbióticos han sido sobre simuladores.

## 5.1. Creación de entradas en */proc*

El primer tema en ser abordado fue la desactivación del *Hyper-Threading*. Sin embargo para poder controlar la activación/desactivación, era necesario crear una interfaz que lo permitiese. El sistema de ficheros `/proc` es un sistema de ficheros virtual, que no se encuentra alojado en disco, sino que está en la memoria principal de la máquina. Es utilizado por el sistema para mostrar información y para el ajuste de ciertos parámetros. Para obtener más información sobre la jerarquía del sistema de ficheros de Linux se puede consultar [30].

Se creó en el directorio `/proc` una entrada denominada `hypertreading` en la que podemos escribir `enable` o `disable` para activar o desactivar el *Hyper-Threading*. Por supuesto, dicha entrada solamente se crea si el procesador en el que está instalado el *kernel* soporta *Hyper-Threading*. Con esto ya queda resuelto el tema de la interfaz.

La creación de entradas en el *procfs* consiste en añadir un *v-nodo* al sistema de ficheros, de forma que cuando alguien lea o escriba en el fichero asociado a nuestro nodo, se llamará a las funciones de lectura y escritura que hemos implementado. Dichas funciones tienen los siguiente prototipos:

```
int (*read)( char* page, char **start, off_t off, int count, int *eof, void *data )
int (*write)( struct file *file, const char __user *buffer, unsigned long count, void *data )
```

En la llamada de lectura los parametros son:

**page** : una página de memoria en espacio de usuario que es en la que hay que escribir la información que queramos devolver al usuario.

**start** : se utiliza para poder realizar lecturas de la entrada en varios accesos. Por ejemplo, si necesitásemos devolver más de los 4k que ocupa una página de memoria, tendríamos que devolver los cuatro primeros kilobytes y dejar el puntero `*start` en el punto en el que nos quedamos leyendo.

**off** : es el punto, dentro de la página, a partir del cual debemos empezar a escribir.

**count** : es el número máximo de bytes que debemos devolver.

**eof** : devolveremos 1 en caso de que se haya llegado al final del fichero.

**data** : es información extra que se nos pasa desde el nodo de la entrada.

Y en la llamada a escritura los parametros son:

**file** : un puntero a la estructura del fichero.

**buffer** : memoria del espacio de usuario en la que está la información que están escribiendo en la entrada.

**count** : número de bytes que se pretenden escribir.

**data** : al igual que en la función de lectura, esta información se nos pasa siempre desde el nodo de la entrada en el *procfs*.

Para poder crear la entrada, creamos el fichero `htaboot.c`. Este fichero se ha ido modificando en el transcurso de este proyecto.

### 5.1.1. Implementación

**__init __hta_proc_start( )** Esta función es ejecutada durante el arranque del sistema. Su función es crear la entrada `hyperthreading` en el directorio `/proc` cuando la máquina en la que se encuentra soporta la tecnología *Hyper-Threading*. Para realizar esa comprobación, se mira la variable `cpu_has_ht`. Las entradas en `/proc` se crean con `create_proc_entry( )` cuya forma de uso se muestra en el siguiente fragmento de código. Las entradas son `struct` del tipo `proc_dir_entry`. El campo `data` apunta a un parámetro que se pasará a las funciones de lectura y escritura. `read_proc` apuntará la función de lectura, en este caso `hta_read_proc` que se explicará a continuación, y `write_proc` a la de escritura, que será `hta_write_proc`.

```
int __init __hta_proc_start( void )
{
    printk( "HTA: Building proc entry... " );
        if( cpu_has_ht ) {
            int i;
            for( i = 0; i < NR_CPUS; i++ )
                ss_kthreads[i] = NULL;
            struct proc_dir_entry *entry = create_proc_entry( "Hyper-Threading", 0666,
                    NULL );
            entry->nlink = 1;
            entry->data = (void*) &__ht_enabled;
            entry->read_proc = hta_read_proc;
            entry->write_proc = hta_write_proc;
            printk( "built.\n" );
        } else
            printk( "HT not available.\n" );
    return 0;
}
```

**hta_read_proc( )** Esta función realiza la lectura de la entrada `hyperthreading`, devolviendo su valor que puede ser `enabled` o `disabled`. Cuando se realice una lectura de ese fichero, se estará llamando a esta función.

```
int hta_read_proc( char* page, char **start, off_t off, int count, int *eof,
        void *data )
{
    if( *(int*)data ) {
        int menor = (count < 8)?count:8;
        strncpy( page, "enabled\n", menor );
        *eof = (menor == 8);
        return menor;
    } else {
        int menor = (count < 9)?count:9;
        strncpy( page, "disabled\n", menor );
        *eof = (menor == 9);
        return menor;
    }
}
```

**hta_write_proc( )** Función que realiza la escritura en el fichero `hyperthreading`. En primer lugar comprueba que el valor es correcto, informando en caso contrario. En función del valor activa o desactiva el *Hyper-Threading* llamando a las funciones `__enable_smt( )` y `__disable_smt( )` que se comentarán en la sección 5.2.

```
int hta_write_proc( struct file *file, const char __user *buffer, unsigned long count,
        void *data )
{
    switch( count ) {
        case 6:
            if( strncmp( buffer, "enable", 6 ) == 0 ) {
                __enable_smt();
                *(int*)data = 1;
                return 6;
            }
            break;
        case 7:
            if( strncmp( buffer, "enable\n", 7 ) == 0 ||
                strncmp( buffer, "enable", 7 ) == 0 ) {
                __enable_smt();
                *(int*)data = 1;
                return 7;
            } else if( strncmp( buffer, "disable", 7 ) == 0 ) {
                if( (*(int*)data) != 0 ) {
                    __disable_smt( );
                    *(int*)data = 0;
                }
                return 7;
            }
            break;
        case 8:
            if( strncmp( buffer, "disable\n", 8 ) == 0 ||
                strncmp( buffer, "disable", 8 ) ) {
```

```
                if( (*(int*)data) != 0 ) {
                    __disable_smt( );
                    *(int*)data = 0;
                }
                return 8;
            }
            break;
    }

    printk( "HTA: You're soooo smart!!!\n" );

    return count;
}
```

## 5.2. Desactivación del *Hyper-Threading*

En esta sección se detalla las diferentes líneas que hemos seguido hasta lograr la activación/desactivación "en caliente", ya que la única forma hasta ahora de hacerlo es en el arranque de la máquina desde la BIOS, siendo imposible modificar su estado una vez arrancado. Se ha intentado hacer de varias formas, aunque solo la alternativa presentada en la sección 5.2.3 es viable.

### 5.2.1. Primer intento

Según [1] cuando uno de los procesadores está en *idle* los recursos se reorganizan de forma que el procesador funciona en modo monotarea. Un procesador se dice que está en *idle* cuando no está realizando nada, el planificador envía la instrucción "HALT" de forma que el procesador (en este caso lógico) se para.

Por tanto se puede llegar a la conclusión que si un procesador entra en *idle* y conseguimos que no le llegue ninguna instrucción, el procesador entra en modo monotarea [1]. En este intento por desactivar el *Hyper-Threading* se bloquean las interrupciones del procesador que se quiere desactivar, evitando así que el planificador le envía instrucciones para ejecutar. Este procesador tampoco ejecutará código del planificador, ya que la ejecución de código del planificador se lleva a cabo mediante el envío periódico de interrupciones a los procesadores (ver Capítulo 2) y al estar estas desactivadas, son ignoradas.

#### Implementación

El fichero `htaboot.c`, aparte de las funciones que se explican a continuación, utiliza la función `void __migrate_all_task( int, int )` implemen-

tada en el fichero del planificador (`sched.c`) y que se muestra en último lugar.

**__disable_smt( )** Función que desactiva el *Hyper-Threading* de todas las CPUs lógicas excepto de la CPU 0. Como en estos procesadores lógicos habrá tareas ejecutándose, habrá que migrarlas a la CPU 0 que es la única que se va a mantener activada. La migración se intentó hacer, sin éxito, mediante una función implementada en el planificador (`sched.c`). Para poder migrar las tareas es necesario que lo haga un hilo de núcleo o *kthread* (ver [20]). En la variable `ss_kthread[cpu]` se guardan los identificadores de los *kthreads*. Se puso un *array* para generalizar, aunque la implementación de *Hyper-Threading* actual es de dos procesadores lógicos por cada procesador físico. Por tanto solamente se usarán las posiciones 0 y 1, correspondientes a cada uno de los procesadores.

Para tratar todas las CPUs que tiene el sistema, utiliza `for_each_cpu( )` de forma que recorre la máscara que se le pasa y hace lo que tenga dentro del bloque de instrucciones con la CPU cuya posición se encuentre a 1. Como la CPU 0 la queremos eliminar de la máscara, utilizamos `cpu_clear( int, cpu_mask )` que borra de una máscara de CPUs la CPU que se pasa como primer parámetro. Las máscaras seleccionan las CPUs poniendo su correspondiente posición a 1, por tanto `cpu_clear( 0, sibling_map )` pondrá a 0 la posición 0. Después para la CPU 1 crea un *kthread* que migra sus tareas a la 0. El código del *kthread* y lo que hace se verá más adelante.

```
void __disable_smt( void )
{
    int this_cpu, cpu;
    this_cpu = smp_processor_id();
    cpumask_t sibling_map = cpu_sibling_map[this_cpu];
    cpu_clear( 0, sibling_map );
    struct task_struct* k;
    for_each_cpu_mask( cpu, sibling_map ) {
        k = kthread_create( stop_sibling_kthread, (void*)this_cpu, "halt/%d", cpu );
        if( !IS_ERR( k ) ) {
            kthread_bind( k, cpu );
            wake_up_process( k );
            ss_kthreads[cpu] = k;
        } else
            printk( "HTA: Couldn't start stop_sibling_kthread for CPU #%d\n", cpu );
    }
    printk( "HTA: Hyper-Threading disabled\n" );
}
```

**__enable_smt( )** Función que activa el *Hyper-Threading* en una CPU física. Esta es la parte más complicada, ya que despertar a una CPU a la que se le han desactivado las interrupciones no es trivial. Si a un procesador en *idle* se le envía una interrupción este se activará. Hay un tipo de interrupciones, denominadas no enmascarables o NMI que son recibidas siempre por el procesador aunque este tenga el ACPI desactivado. Por tanto utilizamos estas para despertar al procesador lógico 1. Esta interrupción la puede enviar la CPU 0, ya que se pueden enviar interrupciones entre los procesadores, denominadas IPIs (*Inter Processor Interrupt*). Para ello hay que preparar la NMI IPI que se va a enviar, y después enviarla. Esto hay que hacerlo tal y como se muestra en el siguiente código. Hay que coger el número de CPU del procesador lógico y seleccionar qué mandar. Los envíos en realidad son escrituras que se realizan con `apic_write_around( )`.

```
void __enable_smt( void )
{
    int i;
    unsigned long cfg, flags;
    for( i = 0; i < NR_CPUS; i++ ) {
        printk( "HTA: Need to stop halt/%d?... ", i );
            if( ss_kthreads[i] != NULL ) {
                printk( "yes.\n" );
                printk( "HTA: Stopping halt/%d... \n", i );
                cpu_set( i, cpu_online_map );
                local_irq_save( flags );
                apic_wait_icr_idle();
                cfg = SET_APIC_DEST_FIELD( cpu_to_logical_apicid(i) );
                apic_write_around(APIC_ICR2, cfg);

                // Prepare the NMI IPI
                cfg = APIC_DM_NMI | APIC_DEST_LOGICAL;
                // Send the NMI IPI
                apic_write_around(APIC_ICR, cfg);
                local_irq_restore( flags );
                ss_kthreads[i] = NULL;
                printk( "HTA: halt/%d stopped.\n", i );
            } else {
                printk( "no.\n" );
            }
    }
    printk( "HTA: Hyper-Threading enabled\n" );
}
```

**stop_siblig_kthread( )** Código del hilo de núcleo que migrará las tareas y parará la CPU en la que se ejecutará. Lo primero que haces es desactivar las interrupciones locales del procesador con `local_irq_save( )` que también hace una copia del estado de los *flags* en ese momento. La máscara `cpu_online_map` tiene asignado un 0 a las CPUs que no están disponibles, y un 1 a las que sí lo están. Por tanto, como la CPU

no va a estar disponible, pasará a valer 0 su posición, para a continuación hacer una llamada a la función
`__migrate_all_tasks(int, int)` que migrará las tareas de la CPU que se pasa como primer parámetro a la CPU que se pasa como segundo parámetro. A continuación ejecutará la instrucción "hlt" de ensamblador[1] ("HALT") para parar esa CPU, que permanecerá en ese estado hasta que le llegue una interrupción no enmascarable o NMI. Cuando esto suceda, continuará con la ejecución y restaurará las interrupciones.

```
int stop_sibling_kthread( void* data )
{
    unsigned long irq_flags;
    preempt_disable();
    local_irq_save( irq_flags );

    int dest_cpu = (int)data;
    int this_cpu = smp_processor_id();

    cpu_clear( this_cpu, cpu_online_map );

    __migrate_all_tasks( this_cpu, dest_cpu );

    __asm__("hlt");
    preempt_enable_no_resched();

    local_irq_restore( irq_flags );
    return 0;
}
```

**__migrate_all_tasks( )** Esta función migra todas las tareas de una CPU de origen a otra de destino. Las CPUs se identifican como `src_cpu` y `dst_cpu` como las CPUs de origen y destino respectivamente. Su funcionamiento consiste en recorrer los distintos *array*s de prioridades (ver 2.1.3). Para pasar una tarea de la cola de una CPU a la cola de la otra, las colas deben estar bloqueadas. La tarea actual no la moverá, ya que se trata del propio hilo de migración, con lo que se la saltará. El código es el siguiente:

```
void __migrate_all_tasks( int src_cpu, int dst_cpu )
{
    runqueue_t *src_rq, *dst_rq;

    src_rq = cpu_rq( src_cpu );
    dst_rq = cpu_rq( dst_cpu );
    double_rq_lock( src_rq, dst_rq );

    int arr, pr;
```

[1] En el *kernel* se utiliza ensamblador en línea mediante *__asm__*. Una muestra de su uso es la instrucción *__asm__("hlt")* de la función *int stop_sibling_kthread(...)*

```
    task_t* task;
    struct list_head* list;
    for( arr = 0; arr < 2; arr++ ) {
        for( pr = 0; pr < MAX_PRIO; pr++ ) {
            list = &src_rq->arrays[arr].queue[pr];
            int requeueus = 0;
            while( !list_empty( list ) ) {
                task = list_entry( list->next, task_t, run_list );
                if( task == current ) {
                    requeueus = 1;
                    list_del( &task->run_list );
                    continue;
                }

                cpu_set( dst_cpu, task->cpus_allowed );
                //Set new CPU to destination CPU
                set_task_cpu( task, dst_cpu );
                task->timestamp = task->timestamp - src_rq->timestamp_last_tick
                    + dst_rq-> timestamp_last_tick;

                // Dequeue task from source runqueue
                src_rq->nr_running--;
                dequeue_task( task, task->array );
                task->array = NULL;

                // Add task to destination runqueue
                activate_task( task, dst_rq, 0 );
            }

            if( requeueus )
                list_add( &current->run_list, list );
            } // for( pr = 0; pr < MAX_PRIO; pr++ )
        } // for( arr = 0; arr < 2; arr++ )
    double_rq_unlock( src_rq, dst_rq );
}
#else
void __migrate_all_task( int src_cpu, int dst_cpu ) {}
#endif
```

### Problemas y conclusiones

Los problemas que encontramos durante el desarrollo de esta solución y los que quedaron sin resolver son numerosos:

1. Al apagar el sistema con el *Hyper-Threading* desactivado se producen *kernel panics* ya que se envían interrupciones al procesador 1 y éste las tiene desactivadas.

2. Durante el proceso de depuración, según [20] la función `printk` funciona en cualquier contexto. Esto no es cierto ya que afecta al funcionamiento del resto del código si es utilizada dentro de un `for_each_cpu( )`.

3. En las colas de las CPUs no solamente hay tareas sino que hay también *kthreads* del sistema. Como se hace una migración indiscriminada, estos *kthreads* pertenecientes a la CPU 1 se quedan en la 0 ya que al activar

el *Hyper-Threading* no se vuelven a migrar a su CPU de origen. Esto provoca una desactivación total del *Hyper-Threading* pero el sistema deja de funcionar correctamente. Por supuesto, al apagar el sistema también se producen problemas al no tener dicho hilos de núcleo en la CPU correspondiente.

4. La CPU 0 no se puede desactivar. Según [1] los dos procesadores lógicos son iguales, pero Linux coloca en el procesador 0 los *kthreads* más importantes, con lo que si se bloquea esta CPU el sistema funcionará incorrectamente.

Debido a todos estos problemas, y a que no se recuperaba bien el estado después de volver a activar el *Hyper-Threading*, se pensó en una solución menos agresiva que la implementada, de forma que no fuese necesario desactivar las interrupciones del procesador lógico 1, sino que para hacer que esté en *idle* hay que evitar que le lleguen tareas nuevas, con lo que habrá que modificar a fondo el planificador. Esta solución pareció más adecuada, ya que el objetivo del proyecto es la modificación del planificador, y con este método se modifican varias de sus funciones. En las siguientes secciones se describe su desarrollo.

### 5.2.2. Segundo intento

En esta nueva rama de desarrollo, se hace una migración de tareas desde la CPU 1 a la 0 cada vez que se desactiva el *Hyper-Threading* pero distinguiendo entre las tareas y los *kthreads*. Según la definición de *kthread* proporcionada por [20] se llegó a la conclusión que un *kthread* no debería tener memoria virtual asignada. Por tanto, en `task` el campo `mm` debe apuntar a `NULL`. Efectivamente este es el método correcto para distinguir los *kthreads* del resto de tareas, con lo que ahora se puede realizar una migración pero dejando en la CPU 1 sus hilos de núcleo para no desestabilizar el sistema. Los hilos que se quedan en la CPU 1, pueden llegar a ejecutar código, pero su tiempo de ejecución es despreciable en situaciones reales.

Otro de los problemas era el manejo de las interrupciones. A partir de ahora, se evita la entrada de nuevos trabajos realizando otra planifiación. Es decir en todas las funciones de `sched.c` encargadas del equilibrado de carga, se evita que este se realice. También hay que tener en cuenta el problema de las tareas nuevas, que deben ir siempre a la CPU 0. Por último, se vió que las tareas que se encuentran suspendidas en el momento de la desactivación,

al despertarse vuelven a su CPU, con lo que se tuvo que tener en cuenta este caso también.

Otro problema añadido son las *affinities* de las tareas. Si una tarea solo se puede ejecutar en el procesador lógico 1 y se desactiva el *Hyper-Threading*, el planificador tal y como estaba no permitiría la ejecución de la tarea en la CPU 0. Este es un problema complejo y que puede tener varias soluciones. Se ha optado por permitir la ejecución de estas tareas en el procesador 0 aunque la tarea no sea afín con este, ya que ha parecido mejor solución que el no permitir que la tarea se ejecute. Por supuesto, el *affinity* de la tarea *no* se modifica, se modifica el planificador para que cuando se den estas circunstancias, se planifiquen las tareas en la CPU 0.

### Implementación

La modificación del planificador ha sido compleja. Sobretodo se han modificado los equilibradores de carga (ver sección 2.3 del Capítulo 2). A continuación ilustraremos lo más relevante:

**get_affinity( )** La función `get_affinity( )` ha sido añadida al planificador para solucionar el problema de las *affinities*. Se explica en primer lugar esta función (aunque fue lo último en hacerse) porque es llamada por varias de las funciones modificadas del planificador. Realiza dos funciones, dice si en la CPU que se pasa como primer parámetro se puede ejecutar la tarea `p`, que es el segundo parámetro, y la máscara entrada/salida dice qué CPUs pueden ejecutar dicha tarea de todas las disponibles.

Si el *Hyper-Threading* está activado (valor de `__ht_enabled` a 1), esta función no hace más que una "AND" de las máscaras `p->allowed` y `cpu_online_map`, que dicen para la tarea p qué CPUs están permitidas y qué CPUs se encuentran en funcionamiento en ese momento respectivamente. La comprobación de si `cpu` puede ejecutar `p` es comprobar que la posición `cpu` de la máscara resultante de la operación "AND" no sea 0. Esto mismo se hacía a lo largo planificador varias veces, con lo que añadiendo y realizando llamadas no se modifica su funcionamiento.

En el caso de tener desactivado el *Hyper-Threading*, habrá que permitir siempre la ejecución en la CPU 0 y no permitirlo en la 1. Para esto, una vez hecha la máscara, se elimina con `cpu_clear(1, *mask)` la

CPU 1 de la máscara, y se permite la 0 con `cpu_set(0, *mask)`. Después comprueba que en la CPU que se pasa como parámetro está permitida la ejecución.

El código de la función es el siguiente:

```
int get_affinity(int cpu, task_t *p, cpumask_t *mask)
{
    runqueue_t *cpu_rq = cpu_rq(cpu);
    struct sched_domain *sd = cpu_rq->sd;

    cpus_and(*mask, sd->span, cpu_online_map);
    cpus_and(*mask, *mask, p->cpus_allowed);

    #ifdef CONFIG_SCHED_HTA
    if( !__ht_enabled && cpu_isset( 1, *mask) ) {
        cpu_set(0, *mask);
        cpu_clear(1, *mask);
    }
    #endif
    if(cpu_isset(cpu, *mask))
        return 1;
    else
        return 0;
}
```

**wake_idle( )** Esta función dice la CPU en la que debería despertarse una tarea. Para equilibrar, esta función devolverá la primera CPU que encuentre en *idle*. Para comprobar si una CPU está en *idle*, utiliza `idle_cpu( cpu )`. Si el *Hyper-Threading* está desactivado, devolvería la 1, cosa que hay que evitar. Para ello se fuerza a que la CPU devuelta sea la 0. El código final es el siguiente:

```
#if defined(ARCH_HAS_SCHED_WAKE_IDLE)
static int wake_idle(int cpu, task_t *p)
{
    #ifdef CONFIG_SCHED_HTA
    if( !__ht_enabled )
        return 0;
    #endif
    cpumask_t tmp;
    runqueue_t *rq = cpu_rq(cpu);
    struct sched_domain *sd;
    int i;

    if (idle_cpu(cpu))
        return cpu;

    sd = rq->sd;
    if (!(sd->flags & SD_WAKE_IDLE))
        return cpu;

    get_affinity(cpu, p, &tmp);
    for_each_cpu_mask(i, tmp) {
    if (idle_cpu(i))
```

```
            return i;
        }

        return cpu;
    }
```

**find_idlest_cpu( )** Esta función encuentra la CPU menos ocupada en ese dominio de planificación. La CPU menos ocupada será aquella que tenga su `runqueue` más corta. En esta función se puede ver un ejemplo de qué fragmento de código es al que sustituye la función `get_affinity( )` explicada anteriormente. Gracias a esta función, no es necesario hacer una distinción en esta función de si el *Hyper-Threading* está o no está activado, ya que de eso se encarga `get_affinity( )`.

```
static int find_idlest_cpu(struct task_struct *p, int this_cpu, struct sched_domain *sd)
{
    int i, min_cpu;
    cpumask_t mask;

    min_cpu = UINT_MAX;
    min_load = ULONG_MAX;

    get_affinity(this_cpu, p, &mask);

    for_each_cpu_mask(i, mask) {
        load = target_load(i);

        if (load < min_load) {
            min_cpu = i;
            min_load = load;

            /* break out early on an idle CPU: */
            if (!min_load)
                break;
        }
    }

    /* add +1 to account for the new task */
    this_load = source_load(this_cpu) + SCHED_LOAD_SCALE;

    /*
     * Would with the addition of the new task to the
     * current CPU there be an imbalance between this
     * CPU and the idlest CPU?
     *
     * Use half of the balancing threshold - new-context is
     * a good opportunity to balance.
     */
    if (min_load*(100 + (sd->imbalance_pct-100)/2) < this_load*100)
        return min_cpu;

    return this_cpu;
}
```

**load_balance( )** A esta función encargada del equilibrado de carga y detallada en el Capítulo 2, se le ha añadido al principio la comprobación de si está o no activado el *Hyper-Threading*[2] . Si está desactivado, como no interesa que haga un equilibrado de carga entre las CPUs, no hace nada.

**load_balance_newidle( )** Esta función es llamada cuando una CPU entra en *idle*, con el fin de que se pasen tareas de las colas de otras CPUs a la suya dentro de ese dominio (ver Capítulo 2). Si se quiere desactivar el *Hyper-Threading* será necesario evitar que cuando la CPU entre en *idle* le incluyan tareas en su cola. Para ello, nada más empezar la función, se añade un fragmento de código idéntico al de la función `load_balance( )`, con la finalidad de que no haga nada. La modificación realizada en el código es como la vista en la nota de pie de página 2.

**idle_balance( )** Esta función la llama `schedlude( )` cuando la CPU que se pasa como primer parámetro va a entrar en *idle*, con lo que le pondrán tareas en su cola para que no esté sin hacer nada. Recorre los dominios de planificación (ver sección 2.1.4 del Capítulo 2) llamando a `load_balance_newidle( )` para asignar a la CPU tareas de otros procesadores de ese dominio. La modificación realizada es como la vista en la nota de pie de página 2.

**active_load_balance( )** Esta función es llamada por los hilos de migración (ver Capítulo 2). Al tener el *Hyper-Threading* desactivado, no interesa que haga un equilibrado, por lo que interesa que no haga nada[3]

**rebalance_tick( )** Es la función de equilibrado que se ejecuta en cualquier CPU cuando llega una señal temporizada del sistema (ver Capítulo 2). Esta señal se trata de una interrupción. En esta función es donde se comprueban las variables `__ht_enabled` y `__ht_disable`, que controlan si el *Hyper-Threading* está o no activado, y en función de sus valores llama a las funciones de activación y desactivación `__enable_smt( )` y `__disable_smt( )` implementadas en el fichero `htaboot.c`

[2]Con el siguiente código se evita que la función realice el equilibrado correspondiente:
#ifdef CONFIG_SCHED_HTA
if( ! __ht_enablesd )
return 0;
#endif

[3]Misma modificación que la mostrada en la nota de pie de página 2.

```
#define CPU_OFFSET(cpu) (HZ * cpu / NR_CPUS)
static void rebalance_tick(int this_cpu, runqueue_t *this_rq,
        enum idle_type idle)
{
    unsigned long old_load, this_load;
    unsigned long j = jiffies + CPU_OFFSET(this_cpu);
    struct sched_domain *sd;

    /* Update our load */
    old_load = this_rq->cpu_load;
    this_load = this_rq->nr_running * SCHED_LOAD_SCALE;
    /*
     * Round up the averaging division if load is increasing. This
     * prevents us from getting stuck on 9 if the load is 10, for
     * example.
    */
    if (this_load > old_load)
        old_load++;
    this_rq->cpu_load = (old_load + this_load) / 2;
    for_each_domain(this_cpu, sd) {
        unsigned long interval = sd->balance_interval;
        if (j - sd->last_balance >= interval) {

#ifdef CONFIG_SCHED_HTA
            if( __ht_enabled && __ht_disable ) {
                if( this_cpu == 1 ) {
                    __disable_smt( );
                }
                sd->last_balance += interval;
                return;
            } else if( !__ht_enabled && !__ht_disable ) {
                sd->last_balance += interval;
                __enable_smt( );
                return;
            }
#endif

            if (load_balance(this_cpu, this_rq, sd, idle)) {
                /* We've pulled tasks over so no longer idle */
                idle = NOT_IDLE;
            }
            sd->last_balance += interval;
        }
    }
}
```

**__migrate_all_tasks( )** La función migra las tareas de una CPU origen a una de destino. En esta implementación, los *kthreads* que se encuentran en la cola de la CPU de origen no se mueven, ya que esto ocasiona problemas. Un *kthread* se diferencia del resto de tareas porque tiene el puntero de memoria virtual a `NULL`. Esta función es llamada desde los hilos de migración implementados en `htaboot.c`.

```
void __migrate_all_tasks( int src_cpu, int dst_cpu )
{
    runqueue_t *src_rq, *dst_rq;
```

```
    src_rq = cpu_rq( src_cpu );
    dst_rq = cpu_rq( dst_cpu );
    double_rq_lock( src_rq, dst_rq );

    int arr, pr;
    task_t* task;

    struct list_head* list;
    LIST_HEAD( kt_list );
    for( arr = 0; arr < 2; arr++ ) {
        for( pr = 0; pr < MAX_PRIO; pr++ ) {
            list = &src_rq->arrays[arr].queue[pr];
            while( !list_empty( list ) ) {
                task = list_entry( list->next, task_t, run_list );
                if( task->mm == NULL ) {
                    list_move_tail( &task->run_list, &kt_list );
                    continue;
                }
                cpu_set( dst_cpu, task->cpus_allowed );
                // Set new CPU to destination CPU
                set_task_cpu( task, dst_cpu );
                task->timestamp = task->timestamp - src_rq->timestamp_last_tick
                + dst_rq-> timestamp_last_tick;

                // Dequeue task from source runqueue
                src_rq->nr_running--;

                dequeue_task( task, task->array );
                task->array = NULL;
                // Add task to destination runqueue
                activate_task( task, dst_rq, 0 );
            }
            while( !list_empty( &kt_list ) )
                list_move_tail( kt_list.next, list );
        } // for( pr = 0; pr < MAX_PRIO; pr++ )
    } // for( arr = 0; arr < 2; arr++ )
    double_rq_unlock( src_rq, dst_rq );
}
```

**try_to_wake_up( )** Esta función intenta despertar una tarea, colocándola en la que considera la mejor CPU. Esta función es bastante compleja, dependiendo de la CPU que ejecuta el código del planificador, de la CPU donde se estaba ejecutando antes y de las afinidades de la tarea. Los cambios introducidos consisten en llamar a `get_affinity( )` para solucionar el problema de las afinidades. Los casos en los que se intenta asignar la tarea a la CPU 1, está tratado en las funciones a las que llama este método, que son las que se han descrito a lo largo de este punto. El código de esta función es excesivamente largo y no se ha añadido a este documento.

En `htaboot.c` las modificaciones son más pequeñas. En los ficheros que se incluyen, se podrá ver que se ha añadido código para tratar la lectura

de contadores hardware. Se empezó a medir bajo esta implementación, pero debido a los problemas encontrados y que se describen en la siguiente sección, no se terminó de implementar el sistema de lectura de los PMCs.

**__disable_smt( )** Es exactamente igual que en el caso anterior. Crea un *kthread* para cada CPU lógica excepto para la 0 y lo lanza en cada una de ellas. El código de estos hilos se explica a continuación.

```
void __disable_smt( void )
{
    __ht_enabled = 0;
    int this_cpu, cpu;
    this_cpu = smp_processor_id();
    cpumask_t sibling_map = cpu_sibling_map[this_cpu];

    cpu_clear( 0, sibling_map );
    struct task_struct* k;
    for_each_cpu_mask( cpu, sibling_map ) {
        k = kthread_create( stop_sibling_kthread, (void*)0, "halt/%d", cpu );
        if( !IS_ERR( k ) ) {
            kthread_bind( k, cpu );
            wake_up_process( k );
        } else {
            __ht_enabled = 1;
            return;
        }
    }
}
```

**stop_sibling_kthread( )** Como se puede ver en el código, la principal diferencia es que esta vez en el código del hilo de migración, no se desactivarán las interrupciones del procesador que se quiere apagar. Migrará todas sus tareas (excepto los *kthreads*), estando la mayor parte del tiempo en *idle* por lo que el *Hyper-Threading* quedará deshabilitado.

```
int stop_sibling_kthread( void* data )
{
    unsigned long irq_flags;
    preempt_disable();

    int dest_cpu = (int)data;
    int this_cpu = smp_processor_id();

    __migrate_all_tasks( this_cpu, dest_cpu );

    preempt_enable_no_resched();

    return 0;
}
```

### Problemas y conclusiones

Esta solución corrige muchos de los problemas que quedaron sin resolver con la anterior implementación. Ya no hay problemas al apagar el sistema,

ya que los *kthreads* del procesador 1 no se mueven de su cola. Esto también hace que el sistema no se vuelva inestable al desactivar y activar el *Hyper-Threading*.

La mayor dificultad ha sido comprender qué es lo que realmente hace cada una de las funciones y como funciona el planificador en conjunto.

Esta implementación se ha ido desarrollando de forma incremental, ya que en un principio no se pensaron todos los casos en los que una tarea podía entrar en ejecución en el procesador lógico 1. Una vez tratados todos los casos y corregidos varios *bugs* detectados, se puede asegurar que si se desactiva el *Hyper-Threading*, asignado `disable` a `/proc/hyperthreading/status` (ver el manual de uso en el apéndice C), ninguna tarea será ejecutada por la CPU 1, por muchas tareas que se lancen y se suspendan.

Después de numerosas pruebas, se detectó que la ejecución del *kthread* encargado de migrar las tareas de la CPU que se desactiva daba problemas. No se llegó a detectar qué produce estos errores ya que todo parecía correcto. La única diferencia aparente entre el *kthread* utilizado para migrar las tareas por la desactivación del *Hyper-Threading* y los hilos de migración utilizados por el planificador para el equilibrado de carga es su creación. En el tercer intento, se modifica justamente esto, para que nuestro hilo se comporte exactamente igual que los hilos de migración.

### 5.2.3. Tercer intento

En esta última solución, el *kthread* se crea en el mismo sitio que los hilos de migración que utiliza el planificador para hacer los equilibrados de carga. Para ello se ha modificado la función `migration_call( )` del planificador, de forma que genere el hilo de migración desde el mismo planificador.

#### Implementación

Las funciones que han cambiado son las siguientes:

**migration_call( )** Función del planificador de Linux cuya función es crear los hilos de migración que permiten equilibrar la carga entre los procesadores. Estos *kthreads* están en todo momento en la CPU en la que son creados. Debido a los problemas descritos en la sección anterior, se decidió hacer exactamente lo que ya estaba hecho y por tanto, comprobado su funcionamiento. Los hilos que permiten la desactivación del

*Hyper-Threading* se crean si se tiene definida la variable de compilación `CONFIG_SCHED_HTA`. El código se puede ver a continuación:

```
/*
 * migration_call - callback that gets triggered when a CPU is added.
 * Here we can start up the necessary migration thread for the new CPU.
 */
static int migration_call(struct notifier_block *nfb, unsigned long action,
        void *hcpu)
{
    int cpu = (long)hcpu;
    struct task_struct *p;
    #ifdef CONFIG_SCHED_HTA
    struct task_struct *m;
    #endif
    struct runqueue *rq;
    unsigned long flags;

    switch (action) {
        case CPU_UP_PREPARE:
            p = kthread_create(migration_thread, hcpu, "migration/%d",cpu);
            if (IS_ERR(p))
                return NOTIFY_BAD;
            p->flags |= PF_NOFREEZE;
            kthread_bind(p, cpu);
            /* Must be high prio: stop_machine expects to yield to it. */
            rq = task_rq_lock(p, &flags);
            __setscheduler(p, SCHED_FIFO, MAX_RT_PRIO-1);
            task_rq_unlock(rq, &flags);
            cpu_rq(cpu)->migration_thread = p;
#ifdef CONFIG_SCHED_HTA
            m = kthread_create(disable_sibling_kthread, hcpu, "disable/%d", cpu );
            if( IS_ERR(m) )
                return NOTIFY_BAD;
            m->flags |= PF_NOFREEZE;
            kthread_bind( m, cpu );
            rq = task_rq_lock( m, &flags );
            __setscheduler( m, SCHED_FIFO, MAX_RT_PRIO-1 );
            task_rq_unlock( rq, &flags );
            cpu_rq(cpu)->disable_rq_thread = m;
#endif
        break;
        case CPU_ONLINE:
            /* Strictly unneccessary, as first user will wake it. */
            wake_up_process(cpu_rq(cpu)->migration_thread);
#ifdef CONFIG_SCHED_HTA
            wake_up_process(cpu_rq(cpu)->disable_rq_thread);
#endif
            break;
#ifdef CONFIG_HOTPLUG_CPU
        case CPU_UP_CANCELED:
            /* Unbind it from offline cpu so it can run.  Fall thru. */
            kthread_bind(cpu_rq(cpu)->migration_thread,smp_processor_id());
            kthread_stop(cpu_rq(cpu)->migration_thread);
            cpu_rq(cpu)->migration_thread = NULL;
#ifdef CONFIG_SCHED_HTA
            kthread_bind( cpu_rq(cpu)->disable_rq_thread, smp_processor_id() );
            kthread_stop( cpu_rq(cpu)->disable_rq_thread );
            cpu_rq(cpu)->disable_rq_thread =  NULL;
#endif
            break;
```

```
        case CPU_DEAD:
            migrate_live_tasks(cpu);
            rq = cpu_rq(cpu);
            kthread_stop(rq->migration_thread);
            rq->migration_thread = NULL;
#ifdef CONFIG_SCHED_HTA
            kthread_stop( rq->disable_rq_thread );
            rq->disable_rq_thread = NULL;
#endif
            /* Idle task back to normal (off runqueue, low prio) */
            rq = task_rq_lock(rq->idle, &flags);
            deactivate_task(rq->idle, rq);
            rq->idle->static_prio = MAX_PRIO;
            _setscheduler(rq->idle, SCHED_NORMAL, 0);
            migrate_dead_tasks(cpu);
            task_rq_unlock(rq, &flags);
            migrate_nr_uninterruptible(rq);
            BUG_ON(rq->nr_running != 0);

            /* No need to migrate the tasks: it was best-effort if
            * they didn't do lock_cpu_hotplug().  Just wake up
            * the requestors. */
            spin_lock_irq(&rq->lock);
            while (!list_empty(&rq->migration_queue)) {
                migration_req_t *req;
                req = list_entry(rq->migration_queue.next,
                migration_req_t, list);
                BUG_ON(req->type != REQ_MOVE_TASK);
                list_del_init(&req->list);
                complete(&req->done);
            }
            spin_unlock_irq(&rq->lock);
            break;
#endif
    }
    return NOTIFY_OK;
}
```

**disable_sibling_kthread( )** Este es el código del *kthread*. Esta diseñado para un solo procesador. Su función es migrar las tareas llamando a la función `migration_all_task( )` y sigue la misma idea que los hilos de migración del planificador. El *kthread* permanece en el procesador hasta que se le indica, con `kthread_should_stop( )`, que tiene que terminar. Es ininterrumpible, y funciona de la siguiente manera:

1. Comprueba que no está en el congelador (*refrigerator*, ver [20]).
2. Salva y desactiva las interrupciones del procesador con `spin_unlock_irqrestore( )`.
3. Comprueba si la CPU está *online*. Si no lo estuviese el *kthread* terminaría.
4. El *kthread* sabe si tiene que migrar tareas comprobando el valor del campo `activate_hta_thread`, el cual fue añadido a las colas

(estructura `runqueue_t`). Esta idea ha sido tomada después de estudiar el funcionamiento de los hilos de migración. Ver el capítulo 2 donde se detalla la estructura de las colas y el funcionamiento de los hilos de migración).

5. Migra las tareas desde su CPU a la de destino y asigna 0 al campo `active_hta_thread` para indicar que ya se ha hecho la migración. Se volverá a hacer cuando este valor cambie.

```
static int disable_sibling_kthread(void * data)
{
    runqueue_t *rq;
    int cpu = (long)data;
    int dest_cpu;
    unsigned long flags;
    // XXX For single processor only
    if( cpu == 0 )
        dest_cpu = 1;
    else
        dest_cpu = 0;

    rq = cpu_rq(cpu);
    BUG_ON(rq->disable_rq_thread != current);

    set_current_state(TASK_INTERRUPTIBLE);
    while (!kthread_should_stop()) {
        if (current->flags & PF_FREEZE) {
            refrigerator(PF_FREEZE);
        }
        spin_lock_irqsave( &rq->lock, flags );
        if (cpu_is_offline(cpu)) {
            spin_unlock_irqrestore( &rq->lock, flags );
            goto wait_to_die;
        }

        if( !(rq->activate_hta_thread) ) {
            spin_unlock_irqrestore( &rq->lock, flags );
            schedule( );
            set_current_state( TASK_INTERRUPTIBLE );
            continue;
        }

        __migrate_all_tasks( cpu, dest_cpu );
        rq->activate_hta_thread = 0;

        spin_unlock_irqrestore( &rq->lock, flags );
    }
    __set_current_state(TASK_RUNNING);
    return 0;

wait_to_die:
    /* Wait for kthread_stop */
    set_current_state(TASK_INTERRUPTIBLE);
    while (!kthread_should_stop()) {
        schedule();
        set_current_state(TASK_INTERRUPTIBLE);
    }
    __set_current_state(TASK_RUNNING);
```

```
    return 0;
}
#else
void __migrate_all_tasks( int src_cpu, int dst_cpu ) {}
#endif
```

**try_to_wake_up( ) y wake_idle( )** En esta función se ha corregido un grave error. Con el *Hyper-Threading* desactivado siempre se devolvía la CPU 0. Esto es crítico en el caso de que la tarea fuese un hilo de migración perteneciente a la CPU 1, ya que al devolver la CPU 0 está diciendo dónde despertar ese hilo. Para solucionar el problema se ha modificado el principio de la función `wake_idle( )` quedando como se ve a continuación:

```
static int wake_idle(int cpu, task_t *p)
{
    /* XXX esto se podria quitar si idle_cpu devolviese 0, ya que la
     * llamada a get_affinity devolvera en la mascara tmp solamente la
     * cpu 0, al eliminar la 1 si el ht esta desactivado
     */
#ifdef CONFIG_SCHED_HTA
    if( !__ht_enabled )
        if( p->mm != NULL )
            return 0;
#endif
    return cpu;
}
#endif
```

Lo que hace es devolver la CPU 0 solamente en caso de que *no* se trate de un *kthread* (recordar que estos hilos no tienen memoria virtual, es decir, tienen su campo `mm` al valor`NULL`). En caso de que lo sea, devolverá la CPU que se pasa como parámetro, que será a la que pertenece el *kthread*.

**__disable_smt( )** Ya no es utilizada.

**stop_sibling_kthread( )** Esta función era el código del *kthread* de la segunda solución propuesta. Por tanto ya no se utiliza, al haber quedado sustituida por `__migrate_all_tasks( )` en el planificador y ya comentada.

### Conclusiones y problemas

Esta solución parecía muy estable, pero después de muchas pruebas, se encontraron fallos de sincronización que provocaban que el sistema dejase de responder.

### 5.2.4. Cuarto intento

Una solución más elegante que el crear nuevos hilos de migración para mover las tareas cuando el *Hyper-Threading* se desactiva, es utilizar los hilos de migración (*migration kthreads*) para hacer esta función (ver 2.3.2).

El *kthread* que en las anteriores implementaciones tenía como función migrar las tareas, ahora hará una selección de las tareas que tienen que ser migradas, creando una lista de `migration_rq_t` (peticiones de migración) para que sean atendidas por los hilos de migración que el planificador dispone. Esta es la idea principal sobre la que se ha basado esta solución, y la función `__migrate_all_task( )` es la que ha sufrido la mayor parte de los cambios.

Otra modificación importante, es realizar más comprobaciones de la tarea que se va a migrar. Por ejemplo, ahora no se migrarán tareas muertas. Estas comprobaciones se realizan a lo largo de las funciones del planificador, y se mostrará en el apartado siguiente qué funciones han sido modificadas.

#### Implementación

**get_affinity( )** Se han añadido las comprobaciones antes mencionadas. Para ello se consulta al campo `flags` de la estrcutra `task_t` de los procesos. En concreto, se comprueban los siguientes flags:

- *PF_EXITING*: está saliendo
- *PF_DEAD*: está muerta
- *PF_SIGNALED*: ha terminado por una señal
- *PF_KSWAPD*: es el *kthread kswapd*. Este *kthread* es el encargado de realizar el *swapping*
- *PF_BORROWED_MM*: algún *kthread* está accediendo a la memoria de la tarea

Aparte de los *flags* se comprueba si la tarea no es el proceso *idle*, el *kthread* de migración, o bien cualquier otro *ktread*. Tal vez las comprobaciones sean redundantes, pero el dejar un caso fuera significa fallo del sistema. La función queda de la siguiente manera:

```
int get_affinity(int cpu, task_t *p, cpumask_t *mask)
{
    runqueue_t *cpu_rq = cpu_rq(cpu);
    struct sched_domain *sd = cpu_rq->sd;

    cpus_and(*mask, sd->span, cpu_online_map);
    cpus_and(*mask, *mask, p->cpus_allowed);

#ifdef CONFIG_SCHED_HTA
    if( !__ht_enabled && cpu_isset( 1, *mask) )
        if( !( p == task_rq(p)->idle ||
                    p == task_rq(p)->migration_thread ||
                    p->mm == NULL ||
                    (p->flags & (PF_EXITING | PF_DEAD | PF_SIGNALED | PF_KSWAPD
                         | PF_BORROWED_MM)) ) ) {
            cpu_set(0, *mask);
            cpu_clear(1, *mask);
        }
#endif
    if(cpu_isset(cpu, *mask))
        return 1;
    else
        return 0;
}
```

**wake_idle( )** Se ha modificado de la misma forma que la función anterior, comprobando el campo `flags`. Las modificaciones han sido las siguientes:

```
static inline int wake_idle(int cpu, task_t *p)
{
#ifdef CONFIG_SCHED_HTA
    if( !__ht_enabled )
        if( !( p == task_rq(p)->idle ||
                    p == task_rq(p)->migration_thread ||
                    p->mm == NULL ||
                    (p->flags & (PF_EXITING | PF_DEAD | PF_SIGNALED | PF_KSWAPD
                         | PF_BORROWED_MM)) ) )
            return 0;
#endif
    return cpu;
}
#endif
```

**migration_call( )** Esta función no cambia respecto a la solución del apartado 5.2.3, creando el *kthread* a la vez que los hilos de migración. Los cambios se harán en el código de la función `__migrate_all_task( )` llamada desde el código del *kthread* (`disable_sibling_kthread( )`).

**disable_sibling_kthread( )** Al igual que la anterior, permanece igual, encontrándose los cambios en la función `__migrate_all_task( )`, que es llamada desde aquí.

**__migrate_all_tasks( )** Esta función ya no mueve tareas, sino que pide al hilo de migración correspondiente que lo haga, mediante una lista de peticiones de migración (*migration_req_t*). En esta lista meterá todas las tareas que cumplan las condiciones que ya se han mencionado a lo largo de esta sección. Después pone los procesos de esta lista en la cola de migración y espera hasta que la migración se realice. Esto es importante para evitar situaciones de carrera.

```
void __migrate_all_tasks( )
{
    migration_req_t migrations[128];
    int n_migrations = 0;
    unsigned long flags;

    runqueue_t *rq = cpu_rq( 1 );
    spin_lock_irqsave( &rq->lock, flags );

    if( rq->nr_running == 0  ||
            (rq->nr_running == 1 && rq->curr == current) ) {
        spin_unlock_irqrestore( &rq->lock, flags );
        return;
    }

    int arr, pr, i;
    task_t* task;

    struct list_head* list;
    struct list_head* begin;
    for( arr = 0; arr < 2; arr++ ) {
        for( pr = 0; pr < MAX_PRIO; pr++ ) {
            list = &rq->arrays[arr].queue[pr];
            begin = list;
            while( begin != list->next ) {
                task = list_entry( list->next, task_t, run_list );
                list = list->next;
                if( task == rq->idle ||
                        task == rq->migration_thread ||
                        task->mm == NULL ||
                        (task->flags & (PF_EXITING | PF_DEAD | PF_SIGNALED |
                                PF_KSWAPD | PF_BORROWED_MM)) )
                    continue;
                init_completion( &migrations[n_migrations].done );
                migrations[n_migrations].type = REQ_MOVE_TASK;
                migrations[n_migrations].task = task;
                migrations[n_migrations].dest_cpu = 0;
                list_add( &migrations[n_migrations].list, &rq->migration_queue );
                n_migrations++;
            }
        } // for( pr = 0; pr < MAX_PRIO; pr++ )
    } // for( arr = 0; arr < 2; arr++ )
    struct task_struct *mt = rq->migration_thread;
    get_task_struct(mt);
    spin_unlock_irqrestore( &rq->lock, flags );
    wake_up_process(mt);
    put_task_struct(mt);
    for( i = 0; i < n_migrations; i++ )
        wait_for_completion( &migrations[i].done );
}
```

**Conclusiones**

Esta es la última implementación desarrollada y que parece solucionar todos los problemas encontrados en las anteriores. Sin embargo, la planificación y sincronización de tareas es sumamente complicada, con lo que puede que no esté exenta de errores.

Esta implementación, con el cambio de estrategia que se ha seguido, ha sido motivado por varios errores de accesos a memoria y sincronización que se han encontrado, con lo que ahora el planificador es más cuidadoso a la hora de mover tareas, esperando a que las migraciones se completen para evitar condiciones de carrera.

## 5.3. Lectura de los PMCs

La lectura de los diversos contadores de rendimiento se ha llevado a cabo desde `htaboot.c`. El procedimiento sigue las pautas descritas en las secciones anteriores, como se verá a continuación. Las modificaciones realizadas al planificador permiten que este funcione de una manera "inteligente", sabiendo en todo momento lo que está pasando en el microprocesador mediante la lectura de los contadores hardware, gracias a los cuales se ha conseguido saber la tasa de fallos de cache, y en función de esta medida, tomar decisiones sobre la activación y desactivación del *Hyper-Threading*. También gracias al uso de estos contadores, se pueden sacar una serie de estadísticas de lo que está pasando el procesador.

Para calcular la tasa de fallos de cache de nivel 1 (L1), es necesario saber el número de fallos de cache producidos y el número de instrucciones *loads* ejecutadas. La tasa de fallos de cache será el resultado de dividir el número total de fallos entre el número de *loads*. Para poder contabilizar estas dos medidas, es necesario contar tres eventos diferentes:

- Fallos de cache de nivel 1 (L1).
- Etiquetado de las instrucciones *load* que se lanzan. Este evento es necesario para poder contar después el número de *loads* retirados del ROB.
- Número de *loads* ejecutados.

También se puede planificar en función de la tasa de fallos de L2. Para calcular esta tasa, es necesario saber el número de fallos producido en L2 y el número de accesos a la memoria de segundo nivel. El número de accesos será el número de fallos producidos en la cache L1. Por tanto es necesario medir un evento más, para el cálculo de esta tasa.

La configuración y lectura de los contadores se ha llevado a cabo siguiendo el proceso descrito en puntos anteriores de este capítulo, y por tanto, basado en lo recogido en [26]. El primer paso es elegir los contadores asociados a cada evento. Esto se ha guardado en las constantes PMC_CACHE, PMC_LOAD_TAG y PMC_LOAD. Para conseguir la configuración adecuada de los registros ESCR y CCCR, ha sido de gran ayuda los programas de monitorización de contadores hardware *Perfctr-2.6.13* y *Brink & Abyss* ([29]). Los dos programas permiten la monitorización de contadores hardware, pero el funcionamientos es diferente.

*Perfctr* muestra al usuario el número de eventos producidos durante la ejecución de un cierto programa que se pasa como parámetro. El evento que se quiere contar se pasa también como parámetro, pero con la peculiaridad de que deben pasarse los valores de los registros ESCR, CCCR y el número de contador, en hexadecimal, con lo que el uso de este programa se vuelve un poco tedioso, y más teniendo en cuenta lo difícil que puede llegar a ser conseguir el valor de estos registros a partir de las muchas tablas documentadas en [26] y [28]. Aparte, al estar trabajando con un Pentium 4, hay una serie de parámetros, como la opción de lectura rápida o conteo preciso (PEBS), que también se pasan como parámetros. También permite medir dos eventos simultáneamente, pero si los eventos no son compatibles para ser medidos en un mismo registro no podrá realizarse las medidas.

Por otro lado, *Brink & Abyss* ofrece al usuario una interfaz de alto nivel, en la que este le pasa al programa el evento que quiere medir configurando un fichero XML. A partir de ahí el programa ya realiza la medida, configurando automáticamente los registros necesarios con solo saber el evento. También permite medir varios eventos a la vez, y al contrario que *Perfctr*, al realizar él la configuración de los registros, consigue que eventos incompatibles entre sí sean medidos. *Brink & Abyss* está compuesto por dos módulos, *Brink* que no es más que un *parser* XML, y *Abyss* que es el encargado de realizar las medidas a partir de la salida de *Brink*. La salida de *Brink*, no es más que la configuración necesaria de los registros, es decir, como si fuesen los parámetros que utiliza *Perfctr*. Un inconveniente es que *Brink & Abyss* solamente

funciona con sistemas con núcleo 2.4. Pero con una modificación, se puede dejar de usar el módulo *Abyss*, que es el que requiere un núcleo 2.4, mientras que *Brink* sí funciona con el 2.6, con lo que se puede aprovechar la facilidad con la que genera las configuraciones de los registros.

Una diferencia importante entre estos dos programas, es que *Perfctr* es genérico y puede monitorizar eventos de cualquier procesador, mientras que *Brink & Abyss* es específico para los Pentium 4 y Xeon. De hecho, uno de los desarrolladores de *Brink & Abyss* es el jefe de desarrollo del sistema de contadores hardware de los Pentium 4, Brinkley Sprunt (ver [29]).

A continuación se muestran los valores utilizados para registros y máscaras. Estos han sido obtenidos gracias al uso conjunto de estos dos programas, tal y como se ha comentado.

```
#define ESCR_L1_MISS        0x12000205
#define CCCR_L1_MISS        0x0003B000
#define PMC_L1_MISS         0x80000010
#define ESCR_L2_RD_MISS     0x18020005
#define CCCR_L2_RD_MISS     0x0003F000
#define PMC_L2_RD_MISS      0x80000000
#define ESCR_LOAD_TAG       0x04000405
#define CCCR_LOAD_TAG       0x00035000
#define PMC_LOAD_TAG        0x8000000C
#define ESCR_LOAD           0x10000205
#define CCCR_LOAD           0x0003B000
#define PMC_LOAD            0x8000000E
#define PEBS_ENABLE         0x01000001
#define PEBS_MATRIX         0x1
```

Se han definido los valores correspondientes para el ESCR, CCCR y el contador a utilizar para cada uno de los tres eventos que se van a tratar. Dos de ellos indican el valor de ESCR y de CCCR, siendo el tercero, llamado PMC_XXX, el que indica en qué contador se contará. Los ESCR_L1_MISS, CCCR_L1_MISS y PMC_L1_MISS son los utilizados para medir los fallos de cache de nivel 1 que se producen durante la ejecución de aplicaciones de usuario. Los ESCR_L2_RD_MISS, CCCR_L2_RD_MISS y PMC_L1_RD_MISS son los utilizados para medir los fallos de cache de nivel 2. ESCR_LOAD_TAG, CCCR_LOAD_TAG y PMC_LOAD_TAG están relacionados con el evento de etiquetado de $\mu$ops "LOAD". Y por último, ESCR_LOAD, CCCR_LOAD y PMC_LOAD para contabilizar los "LOAD" retirados del ROB. En todos eventos, se ha configurado para contar en las aplicaciones de usuario de los dos procesadores lógicos, de ahí que el último valor de estos registros sea 5 en hexadecimal, siendo "00000101" en binario, correspondiéndose los 1's con los bits "T1_USR" y "T0_USR" (ver apartado 4.2).

PEBS_ENABLE y PEBS_MATRIX son utilizados para configurar y activar el modo preciso, utilizado en la lectura de los fallos de cache de nivel 1. También se activa el modo de lectura rápido, solo presente en los procesadores Pentium 4 y Xeon. Se recuerda que este modo consiste en leer únicamente los 32 bits menos significativos del contador (el cual consta de 40 bits).

A continuación se explicará todas y cada una de las funciones relacionadas con la lectura y configuración de contadores de rendimiento. Todas estas funciones se encuentran en el fichero `/arch/i386/kernel/htaboot.c`. Las estructuras de datos utilizadas también serán explicadas detalladamente:

**clear_counter( )** Esta función limpia los registros que se van a utilizar para la lectura de los eventos. El código es el siguiente:

```
static void clear_counters( void )
{
    clear_msr_range(0x3F1, 2);

    clear_msr_range(0x3A0, 26);
    if (/* p4_IQ_ESCR_ok */ 1 )
        clear_msr_range(0x3BA, 2);
    clear_msr_range(0x3BC, 3);
    clear_msr_range(0x3C0, 6);
    clear_msr_range(0x3C8, 6);
    clear_msr_range(0x3E0, 2);
    clear_msr_range(MSR_P4_CCCR0, 18);
    clear_msr_range(MSR_P4_PERFCTR0, 18);
}
```

En la instrucción condicional de la función, se comprueba qué modelo de procesador es dentro de la familia de procesadores Pentium 4 y Xeon, y borra ese campo para los procesadores de modelo 1. Las pruebas se han realizado en un Pentium 4 con *core Northwood* correspondiéndose con el modelo 2, por lo que en nuestro caso nunca se borraría.

**clear_msr_range( )** Borra los `n` contadores (valor que se pasa como segundo parámetro) a partir de una base (primer parámetro), haciendo un recorrido lineal.

```
static void clear_msr_range( unsigned int base, unsigned int n)
{
    unsigned int i;
    for( i = 0; i < n; i++ )
        wrmsr( base+i, 0, 0 );
}
```

**__restart_pmcs( )** En esta función se resetean los PMCs. Esta función es llamada cada vez que se hace una lectura de los contadores. Como se está interesado en calcular la tasa de fallos de cache que se producen

en un periodo de tiempo, cada vez que esto es calculado, los contadores deben ser reinicializados para que estén listos para el siguiente periodo de tiempo. La reinicialización se lleva a cabo por partes, es decir, primero los registros y contadores relacionados con el evento que mide los fallos de cache, después el evento de etiquetado de instrucciones *load* y por último, el evento que cuenta el número de instrucciones *load* retiradas del ROB.

Lo primero es hacer un `preempt_disable( )` para evitar que el proceso sea expulsado de la CPU que lo está ejecutando (y así, volver atómica la ejecución). Con la llamada a `clear_counters( )` se borra el contenido de los contadores. A continuación se lleva a cabo la configuración, para ello se utiliza la función `wrmsr( )`. Esta función utiliza la instrucción "WRMSR" ya comentada y que sirve para poder escribir en estos registros (consultar [27]). El primer parámetro de la función indica el destino y el segundo lo que se quiere escribir. Para calcular el destino correcto es necesario sumar a un valor base ya definido (por ejemplo, MSR_P4_ESCR0 y MSR_P4_CCCR0), un desplazamiento que toma un valor u otro en función del evento y del contador utilizado para contar ese evento. Estos desplazamientos son `cccr_val, pmc, escr_val` y `escr_off`. A continuación se activa el PEBS: primero se inicializan los registros relacionados con el conteo de fallos de cache L1. Después en función de qué tasa queramos medir se procederá a reiniciar unos u otros. Si se van a medir la tasa de fallos de L1, (`!target_level || counting`), se resetearán los contadores de los eventos de etiquetado y el de conteo de "LOADs" retirados del ROB. Si por el contrario se está calculando la tasa de fallos de L2 (`target_level || counting`), se limpian los que cuentan el número de fallos de L2. El proceso es exactamente igual para todos los eventos, como se puede ver en el siguiente fragmento de código, en el que solamente cambian las constantes que contienen los valores correspondientes.

```
void __restart_pmcs( void )
{
    unsigned int escr_val, escr_off, cccr_val, pmc;

    preempt_disable( );

    clear_counters( );
    cccr_val = CCCR_L1_MISS;
    pmc = PMC_L1_MISS & P4_MASK_FAST_RDPMC;
    escr_val = ESCR_L1_MISS;
    escr_off = p4_escr_addr( PMC_L1_MISS & ~P4_FAST_RDPMC, cccr_val ) - MSR_P4_ESCR0;
```

```
        /* We program performance monitor counters for counting L1 cache misses */
        wrmsr( MSR_P4_ESCR0 + escr_off, escr_val, 0 );
        wrmsr( MSR_P4_CCCR0 + pmc, cccr_val, 0 );
        wrmsr( MSR_P4_PEBS_ENABLE, PEBS_ENABLE, 0 );
        wrmsr( MSR_P4_PEBS_MATRIX_VERT, PEBS_MATRIX, 0 );

        if( target_level || counting ) {
            /* We program performance monitor counters for counting L2 read cache misess*/
            cccr_val = CCCR_L2_RD_MISS;
            pmc = PMC_L2_RD_MISS & P4_MASK_FAST_RDPMC;
            escr_val = ESCR_L2_RD_MISS;
            escr_off = p4_escr_addr( PMC_L2_RD_MISS & ~P4_FAST_RDPMC, cccr_val )
                    - MSR_P4_ESCR0;

            wrmsr( MSR_P4_ESCR0 + escr_off, escr_val, 0 );
            wrmsr( MSR_P4_CCCR0 + pmc, cccr_val, 0 );
        }

        if( !target_level || counting ) {
            /* Now for tagging load instructions */
            cccr_val = CCCR_LOAD_TAG;
            pmc = PMC_LOAD_TAG & P4_MASK_FAST_RDPMC;
            escr_val = ESCR_LOAD_TAG;
            escr_off = p4_escr_addr( PMC_LOAD_TAG & ~P4_FAST_RDPMC, cccr_val )
                    - MSR_P4_ESCR0;
            wrmsr( MSR_P4_ESCR0 + escr_off, escr_val, 0 );
            wrmsr( MSR_P4_CCCR0 + pmc, cccr_val, 0 );
            /* And acounting tagged load instructions retired */
            cccr_val = CCCR_LOAD;
            pmc = PMC_LOAD & P4_MASK_FAST_RDPMC;
            escr_val = ESCR_LOAD;
            escr_off = p4_escr_addr( PMC_LOAD & ~P4_FAST_RDPMC, cccr_val )
                    - MSR_P4_ESCR0;
            wrmsr( MSR_P4_ESCR0 + escr_off, escr_val, 0 );
            wrmsr( MSR_P4_CCCR0 + pmc, cccr_val, 0 );
        }

        preempt_enable( );

    }
```

**__read_pmcs( )** Esta función es la encargada de leer los contadores hardware. Para ello llama a la función `rdpmc_low( )`, la cual utiliza la instrucción "RDPMC". El primer parámetro de esta función es el contador que se quiere leer, en este caso guardado en las constantes PMC_CACHE y PMC_LOAD. El segundo parámetro es la variable donde se guardará el valor del contador. Una vez leídos, se resetean los contadores llamando a `__restart_pmcs( )` para que la cuenta vuelva a empezar desde 0.

En esta función también establece distinciones de casos, ya que solo se leen los contadores necesarios para proporcionar los datos que se van a mostrar y que ayudarán en la planificación. Así, si se desea calcular la tasa de fallos de L1, como ya se ha dicho, se leerán las $\mu$ops "LOAD"

retiradas guardando el valor en la variable `loads_retired`. Si son los de L2, pues se leerán los fallos de L2 guardándolos en `l2_re_cache_misses`. La variable `counting` es la encargada de activar estas lecturas o desactivarlas, estableciendo si se debe contar o no. Por supuesto, si la cuenta está desactivada, no se procederá a las lecturas de estos eventos y se asigna directamente 0 al valor de las lecturas. Después, en función de lo que se quiere calcular, se asignará a las variables `loads` y `misses` los valores leídos de los contadores. Para terminar se incrementan las variables encargadas de llevar la cuenta para las estadísticas.

```
void __read_pmcs( unsigned int* loads, unsigned int* misses )
{
    unsigned int l1_cache_misses = 0;
    unsigned int loads_retired = 0;
    unsigned int l2_rd_cache_misses = 0;

    rdpmc_low( PMC_L1_MISS, l1_cache_misses );

    if( target_level || counting )
        rdpmc_low( PMC_L2_RD_MISS, l2_rd_cache_misses );
    if( !target_level || counting )
        rdpmc_low( PMC_LOAD, loads_retired );

    if( target_level ) {
        *loads = l1_cache_misses;
        *misses = l2_rd_cache_misses;
    } else {
        *loads = loads_retired;
        *misses = l1_cache_misses;
    }

    if( counting ) {
        counter_loads += (unsigned long long)loads_retired;
        counter_l1_misses += (unsigned long long)l1_cache_misses;
        counter_l2_misses += (unsigned long long)l2_rd_cache_misses;
    }

    __restart_pmcs( );

}
```

**__too_much_cache_misses( )** Esta función establece si se han producido demasiados fallos de cache. Primero hace una llamada a `__read_pmcs( )` para leer los contadores, guardando el número de fallos de cache en `misses` y el número total de accesos a la memoria correspondiente (o "LOAD" retirados o bien fallos de cache de L1) en `loads`. Como desde el núcleo no se pueden realizar operaciones de punto flotante, para calcular el porcentaje se ha multiplicado por 100 el número de fallos para dividir el producto por `loads`. Después, si se sobrepasa un umbral, definido su valor en `MISS_THRESSHOLD`, se devuelve 1. En caso de que no se supere dicho umbral o el número de "LOAD"

sea 0, se devolverá 0. Puede resultar extraño que en una porción de código no se ejecute ni un solo "LOAD", pero se ha comprobado que sí sucede durante el arranque del sistema, lanzando un *kernel panic* y quedando el sistema colgado. Con esta comprobación el problema queda resuelto.

```
int __too_much_cache_misses( void )
{
    unsigned int loads, misses;

    __read_pmcs( &loads, &misses );
    last_cache_access = loads;
    last_cache_misses = misses;

    if( loads == 0 ) {
        last_miss_percent = 0;
        return 0;
    }
    last_miss_percent = (misses*100)/loads;

    if( last_miss_percent > MISS_THRESSHOLD ) {
        last_limit_surpased = last_miss_percent;
        // printk( KERN_DEBUG "HTA: Cache misses: %u%%\n", miss_percet );
        return 1;
    } else
        return 0;

}
```

## 5.4. Interfaz y estadísticas

Para poder visualizar los resultados de las lecturas de los contadores hardware, se ha diseñado una interfaz en el directorio `/proc`, tal y como se ha explicado en la sección 5.1, en la que se añadía una entrada para poder habilitar y deshabilitar el *Hyper-Threading*. Esta vez se han agrupado todas las entradas en un directorio denominado `hyperthreading`. En este directorio, se puede ver información derivada de las lecturas de los contadores, así como otra información de interés, como los parámetros del planificador propuesto. El funcionamiento del interfaz se puede ver en el apéndice C.

### 5.4.1. Implementación

Las funciones necesarias para la creación de las entradas se han añadido al fichero `htaboot.c`. Se nombrarán las funciones añadidas y lo que hacen, prescindiendo del código debido a su excesiva extensión:

- **int proc_counter_read_proc( char*, char **, off_t, int, int *, void *)**: devuelve al usuario el valor de los contadores.

- **int proc_counter_write_proc( struct file *, const char __user *, unsigned long, void *)**: inicia o detiene los contadores, según lo que escriba el usuario.

- **int max_surpass_count_read_proc( char*, char **, off_t, int, int *, void *)**: devuelve al usuario el número máximo de veces seguidas que se ha sobrepasado el límite de fallos de cache, los cuales serán tenidos en cuenta para reactivar el *Hyper-Threading*.

- **int max_surpass_count_write_proc( struct file *, const char __user *, unsigned long, void *)**: cambia el valor que devuelve la función anterior.

- **int statistics_read_proc( char*, char **, off_t, int, int *, void *)**: devuelve al usuario los valores de los contadores internos de estadísticas.

- **int statistics_write_proc( struct file *, const char __user *, unsigned long, void *)**: esta función no hace nada, se añade para completar la interfaz de llamadas al sistema para *procfs*.

- **int target_level_read_proc( char*, char **, off_t, int, int *, void *)**: devuelve al usuario el nivel de cache en el que está fijado el planificador.

- **int target_level_write_proc( struct file *, const char __user *, unsigned long, void *)**: permite cambiar el nivel de cache en el que se centra el planificador.

- **int miss_threshold_read_proc( char*, char **, off_t, int, int *, void *)**: devuelve el número de fallos máximo que se permiten antes de la desactivación del *Hyper-Threading*.

- **int miss_threshold_write_proc( struct file *, const char __user *, unsigned long, void *)**: establece el número máximo de fallos de cache permitidos antes de la desactivación del *Hyper-Threading*.

- **int status_read_proc( char*, char **, off_t, int, int *, void *)**: indica si el *Hyper-Threading* está o no está activado en ese momento.

- **int status_write_proc( struct file *, const char __user *, unsigned long, void *)**: permite forzar la activación o deactivación del *Hyper-Threading*, o dejarlo a discreción del planificador.

### 5.4.2. Uso de la interfaz

Para utilizar la interfaz del `/proc` simplemente hay que escribir o leer de las entradas creadas, con lo que se llamará a las funciones que se han comentado más arriba. Más información sobre esto se puede encontrar en el manual de uso (C).

## 5.5. Integración

Por último se ha integrado las implementaciones de los últimos tres apartados. En la implementación final se desactiva/activa el *Hyper-Threading* tal y como se ha descrito en el apartado 5.2.3 en función de las lecturas de los contadores de rendimiento hardware, cuya lectura se hace como se ha explicado en 5.3. La interfaz y estadísticas (5.4) fueron ampliándose a medida que se necesitaba más información en la etapa de desarrollo.

### 5.5.1. Implementación

La desactivación del *Hyper-Threading* la lleva a cabo el planificador, pero para saber si debe desactivarlo o activarlo hace una llamada a `__should_disable_smt( )` que devolverá un entero indicando lo que debe hacer. En los trozos de código de los apartados anteriores, se a podido ver como esta esta función estaba presente, y sin embargo no se ha comentado. Esta función es la que decide qué hacer en función de las lecturas de los PMCs. Hace uso de la función `too_much_cache_misses( )` que indica si se ha sobrepasado la tasa de fallos de cache según la política seleccionada por el usuario a través de la interfaz.

Como ya se comentó en el apartado 5.4, el planificador propuesto puede estar o no estar activado. En esta función es donde se realiza la comprobación, para modificar el funcionamiento normal del planificador o dejarlo tal y como está. Para ello se comprueba la variable `hta_sched_enable`.

`force_disable` fuerza al planificador a desactivar el *Hyper-Threading*. Se fuerza su desactivación cuando se desactiva a través de la interfaz (ver 5.4). Por tanto si se está forzando su desactivación y está activado (`__ht_enabled`) se desactivará. En caso de que esté activada la cuenta y por tanto la lectura de los contadores hardware (esto depende del valor de `counting`), los leerá tal y como se ha descrito en 5.3.

Si se está forzando la activación con `force_enable`, la función devolverá siempre 0, para que no sea desactivado. Igual que en el caso anterior, si está activada la cuenta, se procederá a la lectura y cálculo de las tasas de fallos del nivel de cache configurado.

A continuación se comprueba si está activada la opción de planificación HTA (`hta_sched_enable`). En caso negativo, sale de la función devolviendo un 0, que indica la no desactivación del *Hyper-Threading*.

Si `hta_sched_enable` está activado activará y desactivará en función de la tasa de fallos. La función `__too_much_cache_misses( )` dice si se han producidos demasiados fallos de cache, en función del umbral establecido y midiendo la tasa de un determinado nivel de cache (ver 5.3 y 5.4). Se ha añadido un contador de "veces que se sobrepasa el umbral" para poder hacer activaciones más suaves, es decir, en un entorno en el que la mayor parte del tiempo el procesador está con el *Hyper-Threading* desactivado, si en un momento puntual debe activarlo para desactivarlo inmediatamente se realizaría trabajo extra y no se ganaría nada con la desactivación. Con la variable `times_limit_surpassed` se consigue que se realice una cierta resistencia a la activación del *Hyper-Threading* para evitar estas situaciones. Cuando se supera el umbral, `times_limit_surpassed` se incrementa en 1, comprobando que no se supere un límite establecido por `max_times_surpass_count`. Este valor puede ser establecido por el usuario (ver C.3.3).

Si está activado y se producen demasiados fallos de cache, la desactivación es inmediata. Si no supera el umbral de fallos de cache y no está activado el *Hyper-Threading*, se decrementa `times_limit_surpassed` siempre que sea mayor que 0. Cuando este valor llegue a 0, se reactivará el *Hyper-Threading*.

El código de la función es el siguiente:

```
int __should_disable_smt( void )
{
    if( force_disable && __ht_enabled ) {
        if( counting ) {
            unsigned int loads, misses;
            __read_pmcs( &loads, &misses );
        }
        return 1;
    }

    if( force_enable ) {
        if( counting ) {
            unsigned int loads, misses;
            __read_pmcs( &loads, &misses );
        }
```

```
        return 0;
    }

    if( __hta_sched_enabled ) {
        if( __too_much_cache_misses( ) ) {
            if( times_limit_surpassed < max_times_surpass_count )
                times_limit_surpassed++;
            if( __ht_enabled )
                return 1;
        } else if( !__ht_enabled ) {
            if( times_limit_surpassed > 0 ) // XXX: redundant
                times_limit_surpassed--;
            if( times_limit_surpassed == 0 )
                __ht_enabled = 1;
        }
    } else if( counting ) {
        unsigned int loads, misses;
        __read_pmcs( &loads, &misses );
    }
    return 0;
}
```

# Capítulo 6

# Resultados

## 6.1. Modelo de consumo de energía

Las medidas de energía que hemos calculado se han hecho en base a los modelos de consumo de Micron [31].

Hemos utilizado las plantillas de *The Micron© System-Power Calculator* para una memoria de tipo *DDR2* a $333Mhz$. El modelo no es exacto porque nuestra memoria es de tipo *DDR* a $400Mhz$, pero las diferencias entre ambas memorias son mínimas en lo que a consumo se refiere, así pues el modelo es bastante preciso.

Teniendo en cuenta que cada linea de cache en el *Pentium IV Northwood* es de 128 bytes y que la *RAM* puede proporcionar 4 bytes por ciclo, obtenemos que son necesarios 42 ciclos para completar la lectura de una linea de cache (en caso de fallo).

Según el modelo, el consumo de potencia por acceso es de $317{,}5mW$, lo que se traduce en un consumo de energía de $0{,}02J$ por cada bloque de memoria que traemos a cache. Con esto ya podemos hacer el cálculo de la energía que ha consumido un proceso en sus accesos a memoria principal.

## 6.2. *Benchmarks*

Se ha elegido diversas combinaciones de test para probar las diferentes situaciones. Para la elección de dichas combinaciones nos hemos basado en las simulaciones utilizadas en [17]. Se han lanzado 2, 3 e incluso 4 procesos

simultáneos, para comprobar los tiempos de ejecución y consumo con el planificador HTAS activado y desactivado. Las combinaciones y las razones de su elección son las siguientes:

**gzip-art** En este caso un proceso operaciones con enteros[1] (gzip) con un IPC medio y que produce pocos fallos de cache, se lanza con uno de punto flotante[2] (art) con un IPC bajo y un número alto de fallos de cache de primer nivel. Se quiere comprobar si gzip es capaz de aprovechar los recursos mientras art espera debido a los accesos a memoria.

**gzip-gcc-wupwise** En este test se lanzan tres procesos: gzip, gcc de enteros con IPC medio y baja tasa de fallos de cache de primer nivel, y wupwise, un test de punto flotante con un IPC alto.

**gzip-twolf-mcf** Tres procesos de enteros: gzip, twolf con un IPC medio-bajo y el nivel de fallos de cache más alto de los tres, mcf con un IPC muy bajo y con muchos fallos de cache. Con esto veremos como se afectan procesos con muy distintos IPCs y tasa de fallos.

**gzip-gcc** Dos procesos de enteros, gzip con IPC medio y gcc con IPC medio y baja tasa de fallos de cache de primer nivel.

**gzip-gzip** En este caso se lanza el gzip dos veces. Pretendemos comprobar si existen sinergia entre ambos debido al entrenamiento de tablas de predicción.

[1] Perteneciente a SPECINT2000
[2] Perteneciente a SPECFP2000

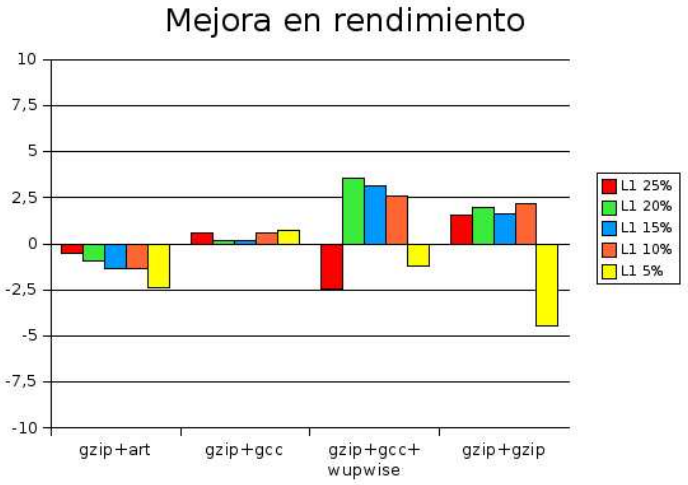


Figura 6.1: Mejora en tiempo de ejecución (planificación por L1) (I)

## 6.3. Rendimiento y consumo

A continuación se mostrarán las medidas obtenidas en la ejecución de las combinaciones de test anteriores. Se ha probado como influye el realizar una planificación teniendo en cuenta la tasa de fallos de cache de nivel 1 y 2. Se ha medido el tiempo de ejecución y se ha obtenido el consumo por los accesos a memoria. Para obtener el consumo se ha utilizado un modelo proporcionado por [31].

### 6.3.1. Umbral basado en L1

En las pruebas realizadas, el umbral para la desactivación del *Hyper-Threading* ha tomado los siguientes valores: 5 %, 10 %, 15 %, 20 % y 25 %. En la figura 6.1, se pueden ver la variación en los tiempos de ejecución:

**gzip-art** Los tiempos de ejecución aumentan a medida que se limita la tasa de fallos de cache, aunque en el peor de los casos el tiempo se incrementa en un 2 %. En este caso, se trata de una de las situaciones ventajosas en las que *Hyper-Treading* resulta efectivo (ver 3.4), ya que los procesos son muy diferentes[3].

**gzip-gcc** Los tiempos de ejecución mejoran ligeramente, al tratarse de procesos similares (ver 3.3).

[3]gzip entero y art de punto flotante e intensivo en memoria

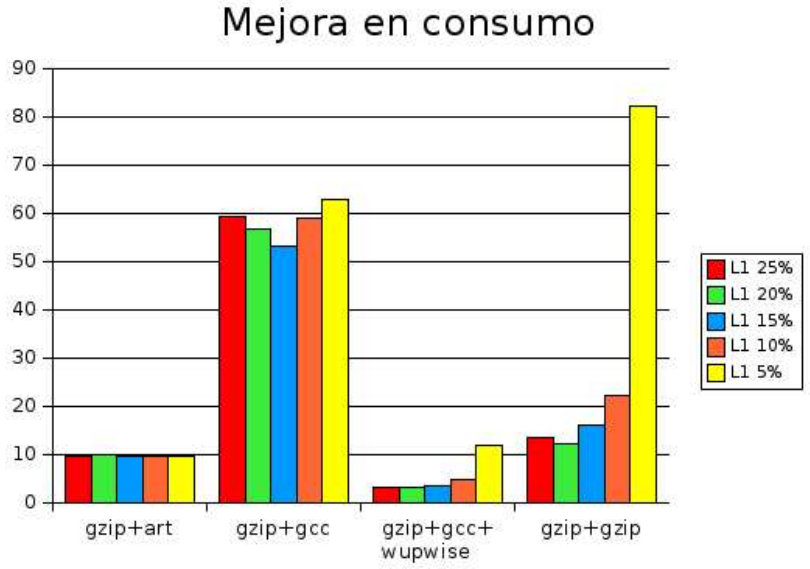


Figura 6.2: Mejora en consumo por accesos de memoria (planificación por L1) (I)

**gzip-gcc-wupwise** Se puede observar como con una limitación del 25 %, el tiempo de ejecución se incrementa entorno al 2.5 %. Sin embargo, si se reduce el límite, el tiempo de ejecución mejora considerablemente, siendo el límite de 20 % el que obtiene unos mejores resultados, entorno al 4 % de mejora. Esta mejora se debe a que al tener dos procesos con una tasa de fallos de cache alta, tienen conflictos por el uso de la cache y el acceso a memoria que limita el rendimiento al no poderse ocultar con *Hyper-Threading*. Al limitar la tasa de fallos y por tanto desactivar el *Hyper-Threading*, estos conflictos desaparecen, aumentando el rendimiento.

**gzip-gzip** Se consigue también mejorar el tiempo de ejecución. Se debe a que como siguen los mismos patrones de acceso, se puede producir competencia, con lo que el funcionamiento con el *Hyper-Threading* desactivado es beneficioso. No se observa el efecto de entrenamiento. Limitando los fallos de cache a 5 % se ve que es contraproducente, ya que no se aprovecha el enmascaramiento de la latencia que proporciona *Hyper-Threading* [1].

Se puede sacar como conclusión, que limitando la tasa de fallos de cache de primer nivel al 20 %, se puede obtener una mejora del renidmiento entorno al 2 %.

En la figura 6.2 se ve como la planificación de HTAS afecta al consumo por accesos a memoria:

**gzip-art** El consumo se reduce en un 10 %.

**gzip-gcc** Es el caso más llamativo, con HTAS activado, se produce una mejore en el consumo del 60 %, respecto al consumo sin HTAS.

**gzip-gcc-wuwise** El consumo mejora entorno al 5 %.

**gzip-gzip** El consumo disminuye a medida que se decrementa el umbral para la tasa de fallos de cache, llegando a consumir en la configuración del 5 % de fallos de cache un 90 % menos que con HTAS desactivado.

En todos los casos el consumo se reduce, siendo esta reducción drástica en algunos casos. Es lógico que esto suceda, ya que se producen muchos menos accesos a la memoria principal debido a la desactivación del *Hyper-Threading*. Al funcionar el procesador en modo monotarea ([1]), los procesos no compiten por la cache y por tanto los accesos a memoria principal disminuyen.

En resumen, una buena configuración para HTAS, consiste en limitar la tasa de fallos de cache de primer nivel al 20 %, ya que como se ha visto, disminuye el consumo por accesos a memoria de manera considerable, sin un deterioro importante del rendimiento, de hecho este aumenta en un 2 %. Con una configuración del umbral al 5 % consigue minimizar el consumo, aunque a costa de perder un 2 % de rendimiento.

Por otro lado, en la figura 6.3 se puede comprobar como procesos que son diferentes[4], con la desactivación del *Hyper-Threading* no se obtienen buenos resultados en cuanto a rendimiento. En cambio, con mgrid-mesa, dos procesos con una caracterización similar, HTAS consigue mejorar el rendimiento, excepto para el caso de tener fijado el umbral al 25 %.

[4]En las pruebas perl-swim, gzip-twolf-mcf y gzip-mcf-equeke-swim (ver B)

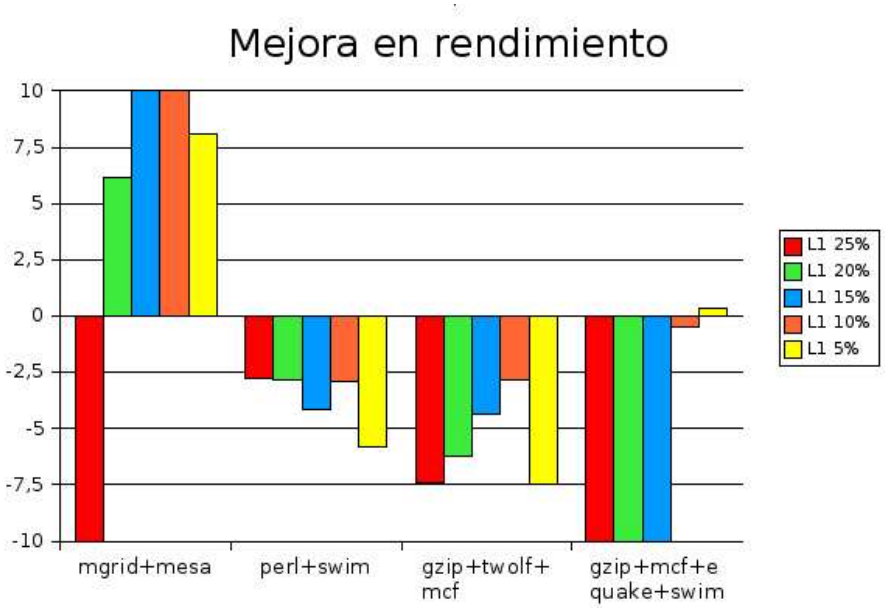


Figura 6.3: Mejora en tiempo de ejecución (planificación por L1) (II)

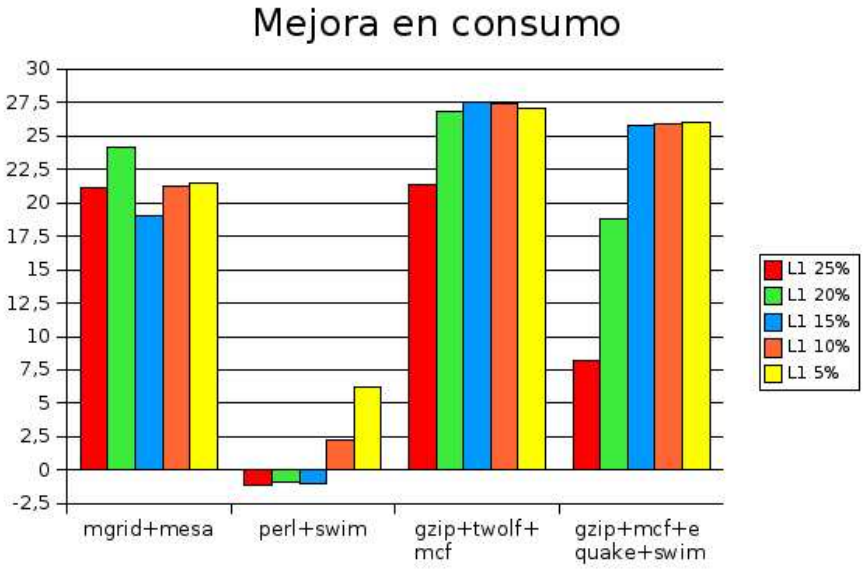


Figura 6.4: Mejora en tiempo de ejecución (planificación por L1) (II)

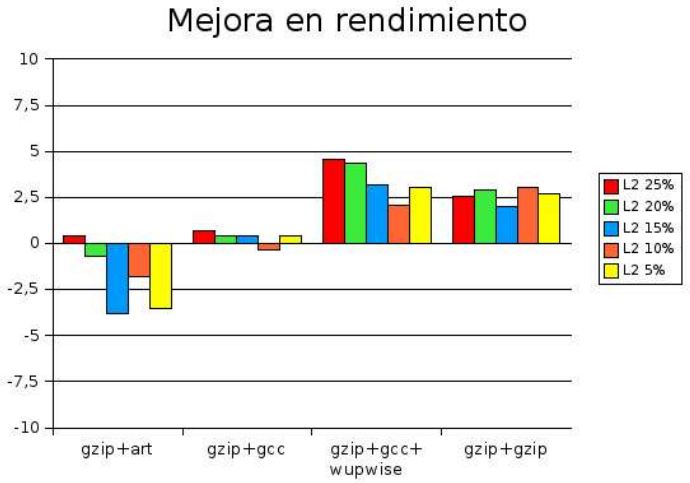


Figura 6.5: Mejora en tiempo de ejecución (planificación por L2) (I)

En cuanto al consumo, en la figura 6.4, como ya se ha visto anteriormente, al limitar los accesos a memoria, se ve disminuido significativamente. El caso especial de estas pruebas, es perl-swim. Estos dos procesos, es la combinación perfecta para aprovechar al máximo la tecnología *Hyper-Threading* (ver 3.4). Uno es un proceso intensivo en CPU de enteros y el otro en punto flotante, con lo entre ellos no hay competencia por las unidades funcionales, por lo que cualquier desactivación de *Hyper-Threading* provoca una bajada en el rendimiento.

### 6.3.2. Planificación respecto a tasa de L2

En la figura 6.5 están los tiempos de ejecución cuando la planificación depende de la tasa de fallos de L2:

**gzip-art** En este caso el rendimiento puede verse afectado ligeramente, dependiendo del caso. Nada significativo teniendo en cuenta la desviación media.

**gzip-gcc** El rendimiento permanece igual

**gzip-gcc-wupwise** El rendimiento aumenta en todos los casos, siendo el caso en el que se limita la tasa de fallos al 25 % y 20 % los que menor tiempo de ejecución consigue, mejorando un 4.5 %.

**gzip-gzip** El tiempo de ejecución disminuye entre un 2 % y un 3 %.

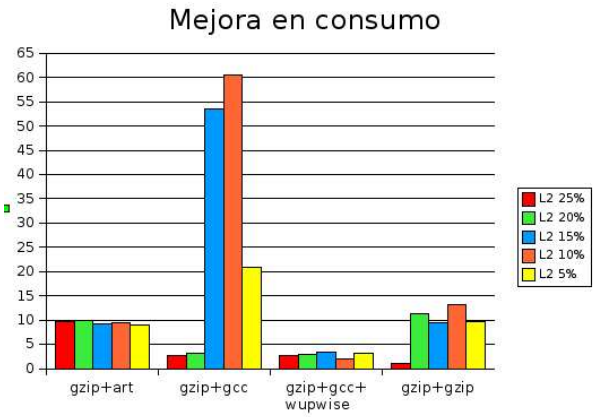


Figura 6.6: Mejora en consumo por accesos a memoria (planificación por L2) (I)

Tratándose de planificación en función a la tasa de L2, se puede decir que el rendimiento general mejora, ya que en el peor de los casos quedaría igual que sin HTAS, mejorando en el resto de los casos. Un valor óptimo es un umbral del 20 %.

Los valores en el consumo por accesos a memoria se ven en la figura 6.6:

**gzip-art** El resultado en esta prueba es rotundo, disminuye para todos los valores del umbral el consumo por accesos a memoria entorno al 10 %.

**gzip-gcc** Mejora para todos los valores, siendo muy significativos para el 15 % y 10 %, en el que el consumo queda en un 40 % de lo consumido sin HTAS.

**gzip-gcc-wupwise** Disminuye ligeramente, aunque puede considerarse que se mantiene debido a errores en las medidas.

**gzip-gzip** Excepto para el caso del umbral al 25 % que tiene igual consumo, para el resto de los valores el consumo disminuye entorno al 10 %.

Si HTAS realiza la planificación en función de la tasa de fallos de L2, el consumo por accesos a memoria se ve mejorado, al igual que el rendimiento en la mayoría de los casos, sobretodo cuando se mezclan procesos con características muy diferentes, como es el caso de gzip-gcc-wupwise.

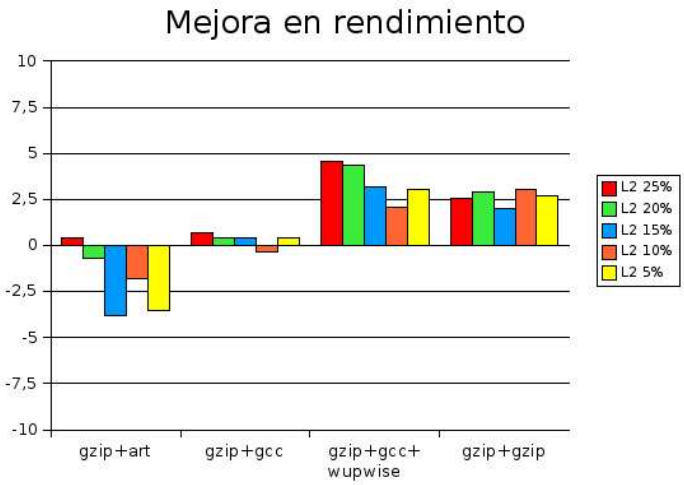


Figura 6.7: Mejora en tiempo de ejecución (planificación por L2) (II)

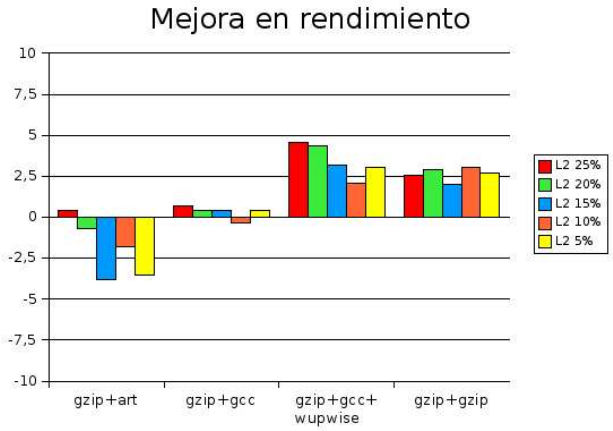


Figura 6.8: Mejora en tiempo de ejecución (planificación por L2) (II)

En la figura 6.7 se pueden ver los resultados de las restantes pruebas. Sucede exactamente lo mismo que cuando se realiza la planificación en función de la tasa de fallos de primer nivel. HTAS únicamente consigue mejorar el rendimiento en el test mgrid-mesa, debido a la similitud de estos procesos. En el resto de los casos empeora el rendimiento, al tratarse de situaciones ventajosas en las que *Hyper-Threading* obtiene buenos resultados.

En cuanto al consumo (figura 6.8) se reduce en todos los casos, aunque en líneas generales en menor medida que cuando se planifica en función de la tasa de L1.

## 6.4. Conclusiones

De los datos analizados en la sección anterior se deduce que HTAS proporciona una mejora en aquellas situaciones en las que los procesos ejecutados son de características parecidas, y que por tanto compiten por los recursos, en este caso por los diferentes niveles de memoria cache. Esta es una de las situaciones en las que *Hyper-Threading* supone una pérdida de rendimiento (3.3), como se ha podido comprobar.

También ha quedado demostrado como el *Hyper-Threading* es capaz de obtener una mejora de rendimiento con procesos muy diferentes, ya que realiza un mejor aprovechamiento de los recursos [1] (ver 3.4), siempre que no se compita por ellos.

Se puede llegar a un compromiso entre rendimiento y consumo, si se desea optimizar considerablemente el consumo, a costa de una ligera bajada de rendimiento, la configuración correcta de HTAS (ver C) será la de planificar en función de la tasa de fallos de primer nivel. Si por el contrario, se quiere mantener el rendimiento, e incluso aumentarlo, se configurará para que planifique en función de la tasa de fallos de la cache de nivel 2, eso sí, el consumo no se reducirá tanto como en el otro caso.

# Capítulo 7

# Conclusiones y trabajo futuro

La tecnología *SMT* está cada vez más extendida. Los procesadores *multicore* con varios *procesadores lógicos* están a punto de aparecer y los fabricantes de procesadores para sistemas empotrados empiezan a incluir *SMT* en sus chips. La importancia del *SMT* en el presente y el futuro próximo está patente.

Ésto hace necesario estudiar las implicaciones que tienen estas tecnologías, y cómo obtener un rendimiento óptimo de ellas.

En este proyecto hemos comprobado que SMT puede ser perjudicial para el rendimiento del sistema y el gran impacto que tiene la política de planificación en la efectividad del SMT. Hemos observado experimentalmente que, incluso con heurísticas que tengan mínimamente en cuenta los recursos compartidos al planificar, ya se consiguen mejoras visibles. Nuestra solución tiene en cuenta los fallos de cache y se pueden ver resultados muy positivos, sobre todo en ahorro de energía.

Entre las posibles mejoras que pueden incluirse en el planificador desarrollado como trabajo futuro citamos las siguientes:

- Utilizar umbrales para la desactivación del *Hyper-Threading* donde no sólo se tenga en cuenta los fallos de cache, sino también otras métricas importantes relacionas con el nivel de ocupación/uso de otros recursos compartidos (unidades funcionales, predictor de saltos...).
- Incluir información de compilación o perfilado (*profiling*) en los ejecutables que puede servir de base al planificador para conocer los recursos que utilizará y guiar la política de asignación de recursos.

- Añadir políticas de calidad del servicio al planificador que tengan en cuenta las peculiaridades de SMT ([17]). Aunque en general, con esta tecnología se consigue mayor productividad, dicha mejora puede conseguirse a expensas de disminuir los recursos disponibles para procesos críticos. Nuestros experimentos preliminares parecen indicar que para conseguir políticas que maximicen el número de plazos cumplidos por tareas con requisitos de tiempo real suave, no basta con la asignación de prioridades tradicional de Linux, sino que es necesario tener en cuenta SMT.

- Un serio problema que nos hemos encontrado a la hora de programar nuestro planificador, ha sido la poca atención que *Intel* ha puesto en el desarrollo e implementación de los contadores hardware. Para implementar planificadores simbióticos, es necesario tener a nuestra disposición un mecanismo de monitorización de eventos hardware mucho más cuidado ([32]). Tal y como están ahora, no permiten cosas tan necesarias como poder contar el número de instrucciones de cada tipo, por ejemplo, tan solo el número de instrucciones de uno o dos tipos. Tampoco permite contar ciertos eventos simultáneamente, o utilizar el *PEBS* (4.1.4) en dos eventos distintos. Sería necesario por lo tanto abordar el diseño de contadores hardware más apropiados.

Para resumir, hemos comprobado que los planificadores simbióticos pueden aportar importantes ganancias y, visto el panorama actual del desarrollo de procesadores, sería imprescindible en un futuro cercano.

# Apéndice A

# Compilación e instalación del *kernel*

En este apéndice se explica descargar, parchear, configurar, compilar e instalar Linux para utilizar el planificador que hemos desarrollado.

## A.1. Descargar Linux

El *site* principal de Linux se encuentra en *http://kernel.org* pero se puede descargar de un *mirror* más cercano, por ejemplo de *rediris* en la siguiente *URL*: *ftp://ftp.rediris.es/pub/linux/kernel/v2.6*

El parche está escrito para Linux *2.6.12.1*, que es la versión más reciente a día de hoy (*25-6-2005*), y el funcionamiento está asegurado para esta versión, pero es muy probable que el parche se aplique limpiamente a versiones anteriores y futuras (dentro de los límites).

Para descargar Linux *2.6.12.1* tendremos que guardar con nuestro gestor de descargas favorito el archivo `linux-2.6.12.1.tar.gz`, por ejemplo. También podríamos descargar la versión comprimida en formato *bzip2*, que tardará menos en descargar pero nos llevará más tiempo descomprimir.

Para descomprimir el núcleo podemos utilizar cualquiera de las herramientas gráficas que ponen a nuestra disposición los entornos de escritorio, o podemos hacerlo desde consola con los siguientes comandos:

Para el formato de fichero *gzip*, sería:

```
prompt:~$ tar xvzf linux-2.6.12.1.tar.gz
```

y para el formato de fichero *bzip2*, sería:

```
\begin{quotation}
        prompt:~$ tar xvjf linux-2.6.12.1.tar.bz2
```

## A.2. Aplicar el parche HTA

Para aplicar el parche utilizaremos el programa `patch`, si el parche está en el mismo directorio en el que se descomprimieron las fuentes de Linux, el fichero con el parche es `patch-2.6.12.1-hta.patch` y el directorio con las fuentes es `linux-2.6.12.1`, el parche se aplicaría de la siguiente forma:

```
prompt:~$ cd linux-2.6.12.1+\newline
prompt:~/linux-2.6.12.1$ patch -p1 < ../patch-2.6.12.1-hta.patch
```

El comando `patch` nos mostrará en su salida dónde y qué partes del parche se han aplicado correctamente.

## A.3. Configurar el *Kernel*

Existen varias interfaces de configuración para el *kernel*, nosotros utilizaremos la interfaz de *ncurses*, para lo cual es necesario tener instaladas en el sistemas las bibliotecas *libncurses* para desarrollo.

Para entrar en la interfaz de configuración gráfica ejecutamos, desde el directorio de fuentes, el siguiente comando:

```
prompt:~/linux-2.6.12.1$ make menuconfig
```

Con esto nos compila y ejecuta el interfaz de configuración con *ncurses*, tras lo cual aparecerá una pantalla como la de la figura A.1.

Si no tenemos mucha experiencia configurando el *kernel*, las opciones que salgan por defecto activadas es más que probable que sean suficientes para nosotros.

Lo primero que tendremos que hacer será entrar en las opciones de *Processor type and features* (figura A.2) para activar las opciones para el parche *Hyper-Threading Aware*.

Para que el parche funcione, deberemos seleccionar como familia de procesadores (*Procressor family*) la del *Pentium 4*, y activar la opción de *HTA (Hyper-Threading Aware) scheduler support* (figura A.3).

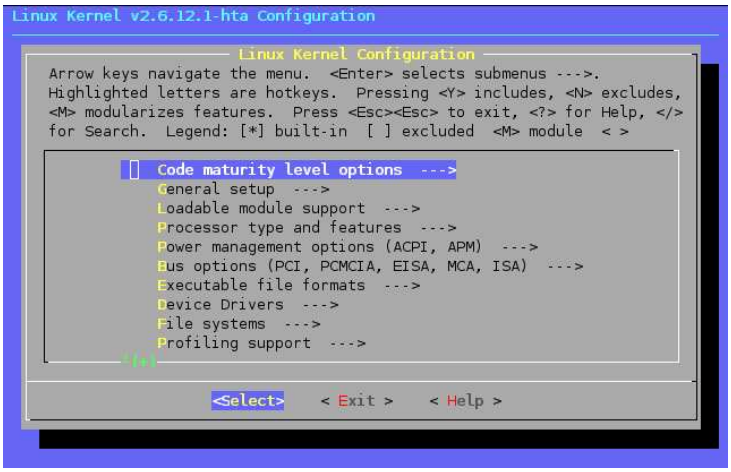


Figura A.1: Interfaz de configuración de Linux con *ncurses*

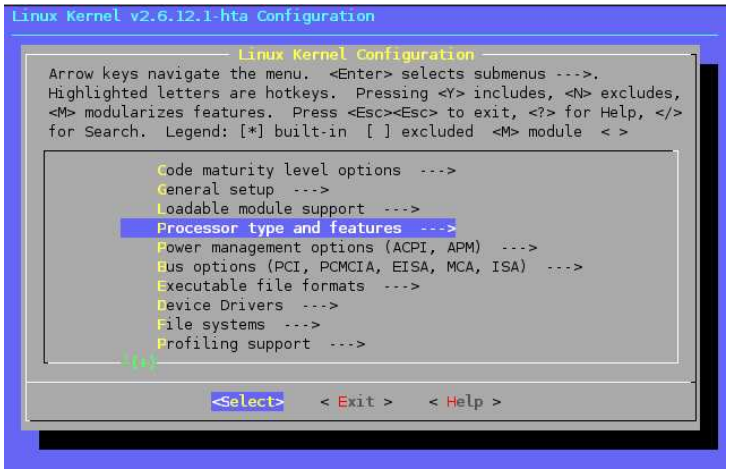


Figura A.2: *Processor type and features*

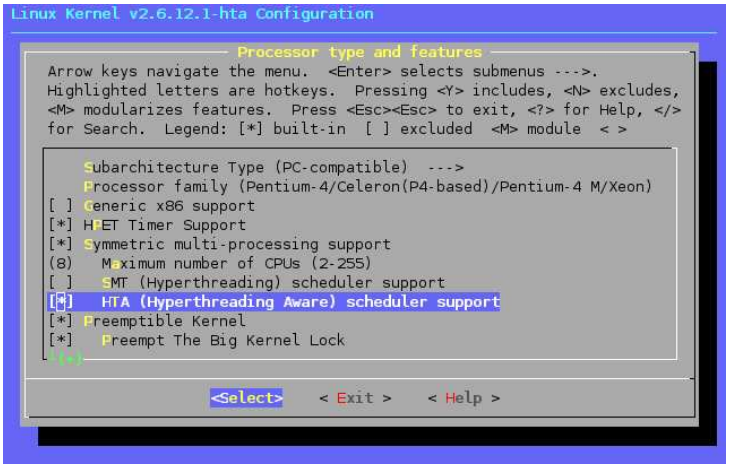


Figura A.3: P4 como tipo de procesador, HTA Scheduler activado

Con esto ya tendremos activado el parche, solo nos resta salir guardando los cambios (figura A.4.

## A.4. Compilar Linux

En esta sección se muestra como se puede compilar Linux por dos métodos diferentes: utilizando el *Makefile* o creando un paquete *Debian*.

El primer método lo podrá utilizar desde cualquier sistema operativo basado en Linux y el segundo le requerirá tener instalado *Debian GNU/Linux*. Aunque el primero es más universal un paquete *Debian* es más fácil de instalar y de desinstalar, así que remendamos éste método si se utiliza *Debian GNU/Linux*.

### A.4.1. Compilación tradicional

Hacer uso de las reglas del *Makefile* para crear el paquete, es tan simple como, desde el directorio de las fuentes, ejecutar:

```
\verb+prompt:~/linux-2.6.12.1$ make
```

Después de un rato (entre 15-25 minutos, según el equipo que utilizado), el *kernel* estará compilado y listo para instalar.

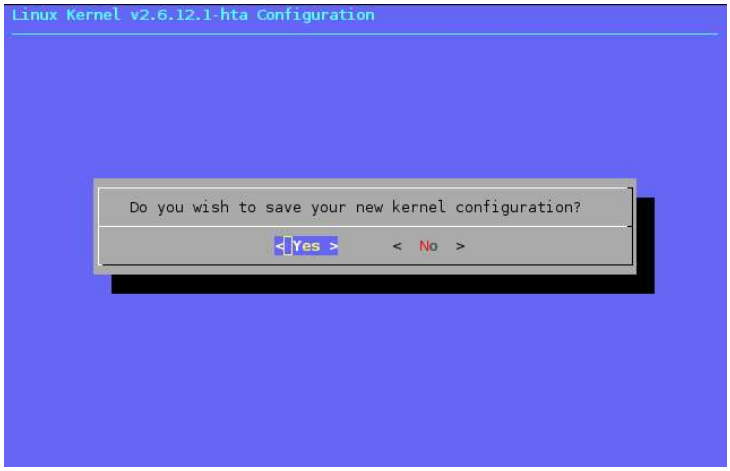


Figura A.4: Guardar los cambios

### A.4.2. Crear un paquete *Debian*

Construir un paquete *Debian* con una imagen del *kernel* tampoco es mucho más complicado, pero requiere la instalación del paquete *kernel-package* que es el que contiene las herramientas que vamos a utilizar. Una vez instalado el paquete, simplemente hay que ejecutar, también desde el directorio de las fuentes, el siguiente comando:

```
prompt:~/linux-2.6.12.1$ make-kpkg kernel_image --revision 1
```

Si no lo ejecutas como *root*, necesitarás añadir una opción suplementaria: `--rootcmd fakeroot`, e instalar el paquete *fakeroot*. Una vez termine la compilación, el nuevo paquete habrá quedado construido en el directorio superior al directorio de las fuentes, con el nombre: *kernel-image-2.6.12.1-hta_1_i386.deb*. Listo para instalar.

## A.5. Instalar Linux

Si se eligió como método de compilado la construcción de un paquete *Debian*, instalar el nuevo núcleo es tan sencillo como ejecutar el el siguiente comando como *root*:

```
prompt:~$ dpkg -i kernel-image-2.6.12.1-hta_1_i386.deb
```

Si se optó por la opción tradicional, la instalación se compone de varios pasos:

1. Instalar los módulos (si los hubiésemos compilado):

   ```
   prompt:~/linux-2.6.12.1$ make modules_install
   ```

2. Copiar la imagen del *kernel* situada en el directorio: `linux-2.6.12.1/arch/i386/boot/bzImage` al directorio de arranque `/boot`

3. Copiar al directorio de arranque el mapa de memoria del *kernel*, que es el fichero `System.map` en el directorio raíz de las fuentes de Linux.

4. Añadir una entrada en el gestor de arranque para el recién instalado *kernel*.

Cómo se añaden entradas al gestor de arranque depende de cual usemos. Para arrancar Linux hay dos que son los más utilizados: *LILO* y *GRUB*. *LILO* fué el primer gestor de arranque que se hizo para Linux, *GRUB* es posterior, más moderno y suple muchas de las carencias de *LILO*.

Si utilizas *GRUB*, añadir una entrada al gestor de arranque es tan simple como ejecutar el comando `update-grub`. *GRUB* detectará el recién añadido núcleo y lo añadirá al menú.

Con *LILO* el proceso puede ser algo más complicado. Hay que añadir una entrada al fichero de configuración en `/etc/lilo.conf` de la siguiente forma:

```
image=/boot/bzImage
        label=linux 2.6.12.1-hta
        read-only
        append="root=LABEL=/"
```

Tras añadir la entrada, es suficiente con ejecutar `lilo` en la linea de comandos para que se grabe el nuevo menú en el *MBR*.

# Apéndice B

# Tablas de rendimiento

En las tablas de esta sección se pueden ver los resultados obtenidos sobre el *benchmark* SPECINT2000 sobre la máquina de pruebas. La máquina sobre la que se han ejecutado los test es:

- Procesador Intel Pentium 4 a 3GHz (*core Northwood*), FSB a 400MHz y 512KB de cache L2.
- 1024MB de memoria principal SDRAM DDR 400MHz

Cada *benchmark* está compilado con gcc y con icc. Se ha medido los ciclos de reloj empleados para la ejecución, el número de fallos de cache de L1 y el IPC conseguido para cada uno de los ejecutables obtenidos. Cada test se ha lanzado en solitario.

## B.1. Benchmarks SPECINT2000

Cuando se ejecuta un mismo test con diferentes parámetros, en la tabla aparece el nombre del test seguido de un número. Para saber qué parámetros se corresponden con cada una de las pruebas hay que consultar las siguientes listas de parámetros:

**gzip**
1. input.source 60
2. input.log 60
3. input.graphic 60
4. input.random 60

5. input.program 60

**vpr** 1. net.in arch.in place.out dum.out -nodisp -place_only -init_t 5 -exit_t 0.005 -alpha_t 0.9412 -inner_num 2
2. net.in arch.in place.in route.out -nodisp -route_only -route_chan_width 15 -pres_fac_mult 2 -acc_fac 1 -first_iter_pres_fac 4 -initial_pres_fac 8

**cc1** 1. integrate.i -o integrate.s
2. scilab.i -o scilab.s
3. expr.i -o expr.s
4. 200.i -o 200.s
5. 166.i -o 166.s

**mcf** 1. inp.in

**crafty** 1. crafty.in

**parser** 1. 2.1.dict -batch

**perlbmk** 1. -I./lib diffmail.pl 2 550 15 24 23 100
2. -I. -I./lib makerand.pl
3. -I./lib perfect.pl b 3 m 4
4. -I./lib splitmail.pl 850 5 19 18 1500
5. -I./lib splitmail.pl 704 12 26 16 836
6. -I./lib splitmail.pl 535 13 25 24 1091
7. -I./lib splitmail.pl 957 12 23 26 1014

**gap** 1. -l ./ -q -m 192M

**vortex** 1. lendian1.raw
2. lendian2.raw
3. lendian3.raw

**bzip2** 1. input.source 58
2. input.program 58
3. input.graphic 58

**twolf** 1. ref

Cuadro B.1: Tabla de SPECINT2000 (1)

| **gzip-1** | | |
|---|---|---|
| Compilador | gcc | icc |
| Ciclos de reloj | 102847504320 | 93945280032 |
| IPC | 0,721047 | 0,65099 |
| Fallos L1 | 3515364036 | 3379132472 |
| **gzip-2** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 46576116200 | 40485948392 |
| IPC | 0,808829 | 0,692441 |
| Fallos L1 | 1340048049 | 1248080832 |
| **gzip-3** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 103169073784 | 95612315064 |
| IPC | 1,02074 | 0,895514 |
| Fallos L1 | 1154427108 | 1165676746 |
| **gzip-4** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 78784460624 | 77050590824 |
| IPC | 0,997484 | 0,860251 |
| Fallos L1 | 1155820458 | 1171159293 |
| **gzip-5** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 186230872752 | 173144797832 |
| IPC | 0,734608 | 0,68343 |
| Fallos L1 | 8179568399 | 7993374663 |

Cuadro B.2: Tabla de SPECINT2000 (2)

| **vpr-1** | | |
|---|---|---|
| Compilador | gcc | icc |
| Ciclos de reloj | 302992113576 | 251850168952 |
| IPC | 0,556383 | 0,629872 |
| Fallos L1 | 3604427864 | 3694352491 |
| **vpr-2** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 293844951520 | 289778239784 |
| IPC | 0,40275 | 0,361341 |
| Fallos L1 | 2976131049 | 3141958165 |
| **cc1-1** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 11985437624 | 9599009648 |
| IPC | 1,07234 | 0,980437 |
| Fallos L1 | 340573855 | 193441232 |
| **cc1-2** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 65973456856 | 59587195440 |
| IPC | 0,942794 | 0,869994 |
| Fallos L1 | 1142008972 | 792226531 |
| **cc1-3** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 11843076944 | 10439099736 |
| IPC | 1,00021 | 0,917427 |
| Fallos L1 | 248420832 | 168823568 |
| **cc1-4** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 113829961232 | 103230381384 |
| IPC | 0,959182 | 0,889891 |
| Fallos L1 | 2048426209 | 1583674750 |

Cuadro B.3: Tabla de SPECINT2000 (3)

| **cc1-5** | | |
|---|---|---|
| Compilador | gcc | icc |
| Ciclos de reloj | 44503861080 | 39227065120 |
| IPC | 0,98259 | 0,760596 |
| Fallos L1 | 1584518819 | 974735040 |
| **mcf** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 585864845392 | 605275878384 |
| IPC | 0,104936 | 0,147645 |
| Fallos L1 | 11248902749 | 11533519372 |
| **crafty** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 328676750904 | 288029600200 |
| IPC | 0,9076 | 0,916819 |
| Fallos L1 | 4122267658 | 3861924298 |
| **parser** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 643647621832 | 697699780648 |
| IPC | 0,809501 | 0,841827 |
| Fallos L1 | 8686669183 | 8803567420 |
| **perlbmk-1** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 47572477056 | 44659234008 |
| IPC | 1,02134 | 1,04577 |
| Fallos L1 | 369363991 | 406397588 |
| **perlbmk-2** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 1885730376 | 1761227864 |
| IPC | 1,08263 | 1,052 |
| Fallos L1 | 2397984 | 4295586 |

Cuadro B.4: Tabla de SPECINT2000 (4)

| **perlbmk-3** | | |
|---|---|---|
| Compilador | gcc | icc |
| Ciclos de reloj | 41310833384 | 40273904936 |
| IPC | 0,791978 | 0,781049 |
| Fallos L1 | 243320685 | 231659207 |
| **perlbmk-4** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 113801483232 | 101633415216 |
| IPC | 1,47151 | 1,4131 |
| Fallos L1 | 556740542 | 457692640 |
| **perlbmk-5** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 63341339464 | 57694293984 |
| IPC | 1,366 | 1,31163 |
| Fallos L1 | 324488898 | 300236165 |
| **perlbmk-6** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 57375630160 | 51367300912 |
| IPC | 1,43366 | 1,3813 |
| Fallos L1 | 287324113 | 247942255 |
| **perlbmk-7** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 104740493704 | 96861003616 |
| IPC | 1,36894 | 1,2936 |
| Fallos L1 | 531859605 | 471096735 |
| **gap** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 270588020488 | 263287407544 |
| IPC | 1,1723 | 1,08479 |
| Fallos L1 | 2541188879 | 2514960336 |

Cuadro B.5: Tabla de SPECINT2000 (5)

| **vortex-1** | | |
|---|---|---|
| Compilador | gcc | icc |
| Ciclos de reloj | 147562266192 | 133371761880 |
| IPC | 1,05178 | 1,05262 |
| Fallos L1 | 1356125417 | 1385594821 |
| **vortex-3** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 166057059736 | 150396077576 |
| IPC | 1,04125 | 1,03835 |
| Fallos L1 | 1530242566 | 1567482594 |
| **bzip2-1** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 160706088160 | 160463588056 |
| IPC | 0,693208 | 0,748845 |
| Fallos L1 | 1229099811 | 1240005146 |
| **bzip2-2** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 161361158368 | 161000091528 |
| IPC | 0,814871 | 0,911129 |
| Fallos L1 | 1059293709 | 1064567898 |
| **bzip2-3** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 204595131568 | 206177040208 |
| IPC | 0,7371 | 0,829514 |
| Fallos L1 | 1176699174 | 1221780736 |
| **twolf** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 891757678544 | 762061468048 |
| IPC | 0,500729 | 0,606359 |
| Fallos L1 | 13786892977 | 14378233939 |

## B.2. Benchmarks SPECFP2000

En esta sección se muestran los resultados de la ejecución de SPECFP2000 (punto flotante) y sus parámetros. Cuando se ejecuta un mismo test con diferentes parámetros, en la tabla aparece el nombre del test seguido de un número. Para saber qué parámetros se corresponden con cada una de las pruebas de SPECFP2000 hay que consultar las siguientes listas de parámetros:

**wupwise** 1. Sin parámetros

**swim** 1. swim.in

**mgrid** 1. mgrid.in

**applu** 1. applu.in

**mese** 1. -frames 1000 -meshfile mesa.in -ppmfile mesa.ppm

**galgel** 1. galgel.in

**art** 1. -scanfile c756hel.in -trainfile1 a10.img -trainfile2 hc.img -stride 2 -startx 110 -starty 200 -endx 160 -endy 240 -objects 10

2. -scanfile c756hel.in -trainfile1 a10.img -trainfile2 hc.img -stride 2 -startx 470 -starty 140 -endx 520 -endy 180 -objects 10

**equake** 1. inp.in

**facerec** 1. ref.in

**ammp** 1. ammp.in

**lucas** 1. lucas2.in

**fma3d** 1. Sin parámetros

**ixtrack** 1. inp.in

**apsi_peak** 1. Sin parámetros

Cuadro B.6: Tabla de SPECFP2000 (1)

| **wupwise** | | |
|---|---|---|
| Compilador | gcc | icc |
| Ciclos de reloj | 452927471704 | 409302831384 |
| IPC | 1,31579 | 1,2992 |
| Fallos L1 | 6043267502 | 6237640063 |
| **swim** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 633568894904 | 483020645960 |
| IPC | 0,600448 | 0,394975 |
| Fallos L1 | 9428071190 | 22272761722 |
| **mgrid** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 743059005704 | 469761563016 |
| IPC | 1,03136 | 0,94231 |
| Fallos L1 | 9042997099 | 7205350885 |
| **applu** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 845687555032 | 572776816088 |
| IPC | 0,86733 | 0,842164 |
| Fallos L1 | 9056140505 | 7233331546 |
| **mesa** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 441759492192 | 410498528288 |
| IPC | 1,03287 | 0,93843 |
| Fallos L1 | 972010789 | 1753339929 |

Cuadro B.7: Tabla de SPECFP2000 (2)

| **galgel** | | |
|---|---|---|
| Compilador | gcc | icc |
| Ciclos de reloj | *N/A* | 392571113280 |
| IPC | *N/A* | 0,839799 |
| Fallos L1 | *N/A* | 19426359432 |
| **art-1** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 695630389000 | 316012449264 |
| IPC | 0,198576 | 0,136536 |
| Fallos L1 | 11779332940 | 5770557949 |
| **art-2** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 711654858632 | 335151635640 |
| IPC | 0,20027 | 0,140403 |
| Fallos L1 | 12059958409 | 6313126341 |
| **equake** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 280741105304 | 253430038184 |
| IPC | 0,735433 | 0,726765 |
| Fallos L1 | 8427546641 | 9245140379 |
| **facerec** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | *N/A* | 389069144144 |
| IPC | *N/A* | 1,09071 |
| Fallos L1 | *N/A* | 4262954643 |
| **ammp** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 1122291688808 | 961141425176 |
| IPC | 0,481187 | 0,417826 |
| Fallos L1 | 11931524447 | 11353640286 |

Cuadro B.8: Tabla de SPECFP2000 (3)

| **lucas** | | |
|---|---|---|
| Compilador | gcc | icc |
| Ciclos de reloj | $N/A$ | 381778414128 |
| IPC | $N/A$ | 0,660701 |
| Fallos L1 | $N/A$ | 5514037947 |
| **fma3d** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | $N/A$ | 533716130832 |
| IPC | $N/A$ | 0,760782 |
| Fallos L1 | $N/A$ | 9065353461 |
| **sixtrack** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 930530554256 | 617958276920 |
| IPC | 1,28557 | 1,05146 |
| Fallos L1 | 1301027420 | 1362334323 |
| **apsi** | | |
| Compilador | gcc | icc |
| Ciclos de reloj | 1218227779000 | 817430914856 |
| IPC | 0,6529 | 0,588447 |
| Fallos L1 | 8026216512 | 8010959853 |

# Apéndice C

# Manual de uso

El planificador *Hyper-Threading Aware* exporta un interfaz al sistema de ficheros *proc*, a través del cual se pueden manipular sus parámetros, ver estadísticas y aprovechar una funcionalidad de lectura de contadores hardware.

Todas las entradas en el *proc* referentes a nuestro planificador están en el directorio `hyperthreading`, así que todas las referencias a las entradas del *proc* son relativas a `/proc/hyperthreading`.

## C.1. Activación y desactivación del planificador

Para controlar la activación y desactivación de nuestro planificador, hemos añadido la entrada `htasched` en el *proc*. Para activar el planificador hay que escribir *enable* en el fichero `htasched`, y para deshabilitarlo se escribe *disable* en `htasched`.

La operación se puede realizar desde consola de la siguiente manera:

```
prompt:/proc/hyperthreading$ echo enable > htasched
prompt:/proc/hyperthreading$ echo disable > htasched
```

Leyendo la entrada `htasched` podemos consultar si está habilitado o deshabilitado el planificador *Hyper-Threading Aware*. La consulta también se puede hacer desde consola de esta forma:

```
prompt:/proc/hyperthreading$ cat htasched
enabled
prompt:/proc/hyperthreading$
```

Hay que tener en cuenta que al desactivar el planificador, el *Hyper-Threading* podría estar deshabilitado. En ese caso habría que rehabilitarlo a mano (C.2) si quisiésemos volver a la planificación tradicional de Linux.

## C.2. Estado del *Hyper-Threading*

También se ha añadido una entrada para consultar si el *Hyper-Threading* está habilitado o deshabilitado, de forma que leer el fichero `status` del *proc* nos devolverá esta información.

La entrada `status` también puede ser utilizada para forzar la habilitación o deshabilitación del *Hyper-Threading*. Escribiendo la palabra `enable` o `disable` en `status`, habilitaremos o deshabilitaremos el *Hyper-Threading*.

Hay que tener en cuenta que lo que se escriba en `status` domina sobre las decisiones que pueda hacer el planificador. Así pues, aunque tengamos activo el planificador *Hyper-Threading Aware*, si hemos forzado la habilitación o deshabilitación del *Hyper-Threading* el planificador no podrá cambiar ese estado.

Si se quiere permitir que el planificador pueda decidir si activar o no el *Hyper-Threading*, es suficiente con escribir `auto` en el fichero `status`.

## C.3. Ajustar los parámetros del planificador

Como se ha explicado en capítulos anteriores, nuestro planificador toma la decisión de activar o desactivar el *Hyper-Threading* según unos parámetros. Dichos parámetros pueden ser consultados y ajustados a través del *proc*.

### C.3.1. Porcentaje de fallos de cache

El límite de fallos de cache a partir de los cuales se desactiva el *Hyper-Threading*, se puede conocer leyendo la entrada `miss_thresshold` del *proc*. Dicho límite viene expresado en fallos por ciento de accesos.

Este límite se puede cambiar escribiendo el nuevo límite máximo en la entrada `miss_thresshold`. Por ejemplo, si quisiésemos cambiar el límite de fallos de cache a un 10 % haríamos lo siguiente:

```
prompt:/proc/hyperthreading$ cat miss_thresshold
20%
prompt:/proc/hyperthreading$ echo 10 > miss_thresshold
prompt:/proc/hyperthreading$ cat miss_thresshold
10%
prompt:/proc/hyperthreading$
```

### C.3.2. Nivel de cache

También podemos establecer el nivel de cache del cual se contarán los fallos. Podemos elegir entre contar fallos de cache de nivel 1 o fallos de cache de nivel 2. El nivel de cache en el que se está fijando el planificador se puede obtener haciendo una lectura de la entrada `target_level`, y puede cambiarse escribiendo el nuevo nivel en `target_level`. Por ejemplo:

```
prompt:/proc/hyperthreading$ cat target_level
L1
prompt:/proc/hyperthreading$ echo 2 > target_level
prompt:/proc/hyperthreading$ cat target_level
L2
prompt:/proc/hyperthreading$
```

### C.3.3. Límite en la cuenta de excesos

Como se ha explicado (5.4), el planificador cuenta el número de veces seguidas que se ha sobrepasado el límite de fallos de cache, de forma que no vuelve a habilitar el *Hyper-Threading* hasta que el número de fallos está por debajo del límite el mismo número de veces. Existe un número máximo de excesos que contamos, tras los cuales no acumulamos más. El valor de ese límite se leer de `max_surpass_count`, y se puede establecer un nuevo límite escribiendo el nuevo valor en `max_surpass_count`.

Cuanto más bajo sea el límite más bruscamente se producirán los cambios de estado por el planificador, cuanto más alto más tenderá el *Hyper-Threading* a permanecer desactivado en situaciones en las que, en general, hay muchos fallos de cache.

## C.4. Estadísticas

Para poder ver qué está pasando con el planificador, se ha añadido una entrada al *proc* con información sobre los últimos sucesos y decisiones tomadas por él. Si se desea acceder a esta información, basta con leer el contenido de `statistics`.

La información que encontraremos es la siguiente:

- *Last cache access*: el número de accesos al nivel de cache seleccionado la última vez que se leyeron los contadores hardware.
- *Last cache misses*: el número de fallos de cache del nivel seleccionado.
- *Last miss percent*: el porcentaje de accesos a cache que fueron fallo.
- *Last limit surpassed*: el porcentaje de fallos de cache que se alcanzó la última vez que se superó el umbral de desactivación.
- *Times limit surpassed*: el número de veces seguidas que se ha superado el umbral hasta el momento de la lectura.

## C.5. Contador

Disponemos de un mecanismo para contar los *load*, fallos de cache de nivel 1 y fallos de cache de nivel 2 entre instantes de tiempo. La entrada `counter` del *proc* nos proporciona esa funcionalidad. Podemos iniciar la cuenta en cualquier momento escribiendo `start` en el fichero y podemos detenerla escribiendo `stop`. Para saber cuantos eventos ha contado solo necesitamos leer el fichero, lo que nos devolverá la siguiente información:

- *Loads*: el número de instrucciones de *load* ejecutadas.
- *L1 misses*: el número de fallos de cache de nivel 1.
- *L2 misses*: el número de fallos de cache de nivel 2.

# Bibliografía


[1] *Hyper-Threading Technology*, Intel Technology Journal, Volume 06 Issue 01, Published February 14 2002, ISSN: 1535766X

[2] *A single-chip multiprocessor*, L. Hammand, B. A. Nayfeh, & K. Olukotum, IEEE Computer, 30(9):79-85,Sep. 1997.

[3] *A Survey of Processors with Explicit Multithreading*, T. Ungerer, B. Robie & J. Silc, ACM Computing Surveys, Vol.35, No.1, March 2003, pp. 29-63

[4] *Simultaneous multithreading: Maximizing on-chip parallelism*, D. M. Tullsen, S. Eggers, & H. M. Levy, In Proc. ISCA-22, 1995.

[5] *Simultaneous Multithreading: A Platform for Next-Generation Processors*, S.J. Eggers, J.E. Emer, H.M. Levy, J.L. Lo, R.S. Stamm, D.M. Tullsen, IEEE Micro, pp.12-19,1997

[6] *Chip Multithreading: Opportunities and Challenges*, L. Spracklen & S.G. Abraham, In proceeding of de 11th International Symposium on High-Performance Computer Architecture (HPCA'05), San Francisco, USA, February 2005, pp.248-252.

[7] *EV8: the post-ultimate Alpha*, J.S. Emer, In International Conference on Parallel Architectures and Compilation Techniques, September 2001.

[8] *Increasing Superscalar Performance throught Multistreaming*, W. Yamamoto, M. Nemirovsky, Proceeding of the International Conference on Parallel Architectures and Compilation Techniques (PACT), 1995, pp.49-58.

[9] *POWER4 System Microarchtecture*, J.M. Tendler, J.S. Dodson, J.S. Fields Jr., H. Le, B. Sinharoy, IBM Journal Research and Development Vol. 46(1), Enero 2002.

[10] *Best Servers of 2004. Where Multicores Is the Norm*, K. Krewel, In-Stat/ MDR-Newsletter, January 18, 2005 Disponible en http://www.mpronline.com

[11] *Montecino - The next product in the Itanium Processor Family*, C. McNairy, R. Bhatia, HotChips 16, 24 August 2004

[12] *Power5: IBM's next generation power microprocessor*, R. Kalla, B. Sinharoy & J. Tendler, In Proc. 15th Hot Chips Symp, pages 292-303, August 2003.

[13] *IBM Power5 chip: A dual-core multithreaded processor*, R. Kalla, B. Sinharoy & J. Tendler, IEEE Micro, March 2004.

[14] *Power Efficiente Processor Architecture and The Cell Processor*, H.P. Hofstee, In Proceeding of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05) San Francisco, USA, February 2005, pp.258-262.

[15] *Thread Coloring: A Scheduler Proposal from User to Hardware Threads*, Marisa Gil, Ruben Pinilla, ACM SIGOPS Operating Systems Review, Vol. 39, No. 2, pp. 54-70, April, 2005.

[16] *Symbiotic Jobscheduling for a Simultaneous Multithreading Processor*, Dean M. Tullsen & Allan Snavely, in Procceding of ASPLOS IX, November 2000.

[17] *QoS for High-Performance SMT Processors in Embedded Systems*, Francisco J. Cazorla, Alex Ramirez y Mateo Valero, Published by the IEEE Computer Society.

[18] *Computer Architecture: A Quantitative Approach (3rd Edition)*, J. Hennessy & D. Patterson, Morgan Kaufmann Publishers Inc, 1999

[19] *Parallel Computer Architecture: A hardware/software approach*, David E. Culler, Jaswinder P. Singh, Morgan Kaufmann Publishers, Inc., 1999

[20] *Linux Kernel Development*, Rovert Love, Ed. Sams Publishing, 2004, ISBN: 0-672-32512-8

[21] *Understanding the Linux 2.6.8.1 CPU Scheduler*, Josh Aas (josh@trancesoftware.com)

©2005 Silicon Graphics, Inc. (SGI)
17th February 2005
http://josh.trancesoftware.com/linux/

[22] Parche *hyperthreading aware scheduler* de *Ingo Molnar* para *Linux 2.5.31-BK-curr*, http://kerneltrap.org/sup/391/HT-aware-scheduler-2.5.31-BK.patch

[23] *Linux on NUMA Systems*, Proceedings of the Linux Symposium, Volume One. July 21th-24th, 2004 Ottawa, Ontario, Canada. http://www.linuxsymposium.org/proceedings/reprints/Reprint-Bligh-OLS2004.pdf

[24] *Scheduling domains*, revista LWN.net. https://lwn.net/Articles/80911/

[25] Linux Kernel 2.6 Features in Red Hat Enterprise Linux. Copyright ©2002 Red Hat, Inc. http://www.redhat.com/whitepapers/rhel/RHELwhitepaperfinal.pdf

[26] *IA-32 Intel Architecture Software Developer's Manual. Volume 3: System Programming Guide*, Order Number: 253668, http://developer.intel.com

[27] *IA-32 Intel Architecture Software Developer's Manual. Volume 2B: Instruction Set Reference, N-Z*, Order Number: 253667i http://developer.intel.com

[28] *IA-32 Intel Architecture Optimization. Reference Manual*, Order Number: 248966-011 http://developer.intel.com

[29] Página oficial del programa *Brink & Abyss*, http://www.eg.bucknell.edu/~bsprunt/emon/brink_abyss/brink_abyss.shtm

[30] Página de documentación de Debian, http://www.debian.org/doc/

[31] The Micron© System-Power Calculator, http://www.micron.com/products/dram/syscalc.html

[32] *Killer EMON Application*, Brinkley Sprunt, in HPCA 11th, San Franciso, February 2005.